GRAPHICS PERFORMANCE BENCHMARKING OF ANDROID DEVICES USING
OPENGL ES

A thesis submitted in partial fulfillment of the requirements
For the degree of Master of Science in
Software Engineering

by

Aram Akopian

May 2015
The thesis of Aram Akopian is approved:

___________________________________                                  ____________________
Prof. G. Michael Barnes

___________________________________                                  ____________________
Prof. Robert D. McIlhenny

___________________________________                                  ____________________
Prof. Richard Covington, Chair

California State University, Northridge
DEDICATION

I dedicate this work to my family. I am very grateful for their support and love. They continuously provide me encouragement and motivation to keep learning, working, and always trying to better myself.

I also dedicate this to my friends. Words cannot express how fortunate I feel to have them in my life. I thank them for always being a positive influence, supporting me, and I wish them a life full of success and opportunity.
ACKNOWLEDGMENT

I would like to express my sincere gratitude to all those who provided their support and guidance.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copyright</td>
<td>ii</td>
</tr>
<tr>
<td>Signatures</td>
<td>iii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgement</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xii</td>
</tr>
<tr>
<td>Abstract</td>
<td>xiii</td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Project Goal</td>
<td>2</td>
</tr>
<tr>
<td>Approach</td>
<td>2</td>
</tr>
<tr>
<td>Android and Mobile Graphics</td>
<td>4</td>
</tr>
<tr>
<td>OpenGL ES 2.0 Overview</td>
<td>9</td>
</tr>
<tr>
<td>Mobile GPU Architecture</td>
<td>13</td>
</tr>
<tr>
<td>Trade Study</td>
<td>16</td>
</tr>
<tr>
<td>Benchmarking Concepts</td>
<td>20</td>
</tr>
<tr>
<td>The Purpose of Benchmarking</td>
<td>20</td>
</tr>
<tr>
<td>Frame Rate</td>
<td>20</td>
</tr>
<tr>
<td>Bottlenecks</td>
<td>22</td>
</tr>
<tr>
<td>V-Sync</td>
<td>23</td>
</tr>
<tr>
<td>Antialiasing</td>
<td>25</td>
</tr>
<tr>
<td>Application Requirements</td>
<td>27</td>
</tr>
<tr>
<td>Application Design</td>
<td>29</td>
</tr>
<tr>
<td>Features</td>
<td>29</td>
</tr>
<tr>
<td>UML Diagrams</td>
<td>30</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>User Interface Design</td>
<td>41</td>
</tr>
<tr>
<td>Benchmark Scene Design</td>
<td>52</td>
</tr>
<tr>
<td>Test Execution Model</td>
<td>53</td>
</tr>
<tr>
<td>Anatomy of a Scene</td>
<td>57</td>
</tr>
<tr>
<td>Project Environment</td>
<td>78</td>
</tr>
<tr>
<td>Device Performance Data</td>
<td>88</td>
</tr>
<tr>
<td>Devices Under Test</td>
<td>88</td>
</tr>
<tr>
<td>Results</td>
<td>90</td>
</tr>
<tr>
<td>Analysis</td>
<td>95</td>
</tr>
<tr>
<td>Future Work</td>
<td>101</td>
</tr>
<tr>
<td>Conclusion</td>
<td>103</td>
</tr>
<tr>
<td>References</td>
<td>107</td>
</tr>
<tr>
<td>Appendix A</td>
<td>109</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 1: Android OS Platform Versions as of March 2, 2015 5
Table 2: OpenGL ES Versions Used on Android as of March 2, 2015 5
Table 3: OpenGL ES Versions Supported by Android 10
Table 4: Hardware Specs for Devices Under Test 88
Table 5: Frame Rate Results for Acer Iconia A1 90
Table 6: Frame Rate Results for Samsung Galaxy Tab 2 91
Table 7: Frame Rate Results for HTC One V 92
Table 8: Frame Rate Results for Samsung Galaxy Note 3 93
Table 9: Frame Rate Results for HTC One 93
Table 10: Frame Rate Results for LG G3 94
Table 11: Frame Rate Comparison – No AA vs. 4x AA 100
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Technology and Architecture</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Android's Surface Rendering Pipeline</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Data Flow in Android's Graphics Pipeline</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>OpenGL ES 2.0 Programmable Graphics Pipeline</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>Immediate Mode Rendering Diagram</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>Tile Based Deferred Rendering Diagram</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>Scene Divided into Tiles for Rendering</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>Frame Rate as Frame Time vs Frames Per Second</td>
<td>22</td>
</tr>
<tr>
<td>9</td>
<td>Example of Antialiased vs. Aliased Edges</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>Use Case for Main Interaction</td>
<td>31</td>
</tr>
<tr>
<td>11</td>
<td>Use Case for Test Customization</td>
<td>32</td>
</tr>
<tr>
<td>12</td>
<td>Package Overview</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>Package Contents - opengl</td>
<td>34</td>
</tr>
<tr>
<td>14</td>
<td>Package Contents - ui</td>
<td>35</td>
</tr>
<tr>
<td>15</td>
<td>Package Contents - model</td>
<td>35</td>
</tr>
<tr>
<td>16</td>
<td>Package Contents - adapter</td>
<td>36</td>
</tr>
<tr>
<td>17</td>
<td>Package Contents - util</td>
<td>36</td>
</tr>
<tr>
<td>18</td>
<td>Sequence Diagram for Running a Test</td>
<td>40</td>
</tr>
<tr>
<td>19</td>
<td>User Interface Mockup</td>
<td>41</td>
</tr>
<tr>
<td>20</td>
<td>MOBILESpeed App Home Screen</td>
<td>46</td>
</tr>
<tr>
<td>21</td>
<td>MOBILESpeed App Tests Screen</td>
<td>47</td>
</tr>
<tr>
<td>22</td>
<td>MOBILESpeed App Results Screen</td>
<td>49</td>
</tr>
<tr>
<td>23</td>
<td>MOBILESpeed App Device Info Screen</td>
<td>50</td>
</tr>
<tr>
<td>24</td>
<td>MOBILESpeed App About Screen</td>
<td>51</td>
</tr>
<tr>
<td>25</td>
<td>Test Execution Model Diagram</td>
<td>53</td>
</tr>
<tr>
<td>26</td>
<td>Rendering Scene of 20,000 Points</td>
<td>62</td>
</tr>
</tbody>
</table>
Figure 27: Rendering Scene of 20,000 Lines 63
Figure 28: Rendering Scene of 10,000 Triangles 64
Figure 29: Rendering Scene of 512x512 Textures 65
Figure 30: Rendering Scene of Textures with Lighting 67
Figure 31: Rendering Scene of Colliding Spheres 69
Figure 32: Rendering Scene of Dynamic Spheres 70
Figure 33: Rendering Scene of Static Frames 72
Figure 34: Process for Benchmarking a Static Scene 74
Figure 35: Rendering Blank Scene to Test Bus Bandwidth 75
Figure 36: Rendering Scene of Combined Elements 76
Figure 37: Creating a New Android Application Project 80
Figure 38: New Android Application Project Details 81
Figure 39: Android Application Configuration - 1 82
Figure 40: Android Application Configuration - 2 82
Figure 41: Android Application Configuration - 3 83
Figure 42: Eclipse Project View 84
Figure 43: How to Run an Application in Eclipse 85
Figure 44: Android Device Chooser in Eclipse 86
Figure 45: Git Installation for Eclipse 87
Figure 46: Integrated Git Toolbar in Eclipse 87
Figure 47: Graph of Test Results 96
Figure 48: Antialiasing Up-close 100
Figure 49: Detailed Classes of the UI Package 109
Figure 50: Detailed Classes of the Model Package 110
Figure 51: Detailed Classes of the Adapter Package 111
Figure 52: Detailed Classes of the Util Package 111
Figure 53: Detailed Classes of the OpenGL Package - Part 1 112
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS</td>
<td>Frames per second</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open graphics library</td>
</tr>
<tr>
<td>OpenGL ES</td>
<td>OpenGL for embedded systems</td>
</tr>
<tr>
<td>GLES</td>
<td>OpenGL for embedded systems</td>
</tr>
<tr>
<td>GLSL</td>
<td>OpenGL Shading Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JDK</td>
<td>Java SE Development Kit</td>
</tr>
<tr>
<td>ADT</td>
<td>Android Development Tools</td>
</tr>
<tr>
<td>MVP</td>
<td>Model View Projection</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>TMU</td>
<td>Texture Mapping Unit</td>
</tr>
<tr>
<td>DPI</td>
<td>Dots Per Inch</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>App</td>
<td>Application</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>V-Sync</td>
<td>Vertical Synchronization</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>IMR</td>
<td>Immediate Mode Renderer</td>
</tr>
<tr>
<td>TBR</td>
<td>Tile-Based Renderer</td>
</tr>
<tr>
<td>TBDR</td>
<td>Tile-Based Deferred Rendering</td>
</tr>
</tbody>
</table>
ABSTRACT

GRAPHICS PERFORMANCE BENCHMARKING OF ANDROID DEVICES USING OPENGL ES

By

Aram Akopian

Master of Science in Software Engineering

As the variety of mobile Android devices available on the market continues to increase and the popularity of mobile games and visually impressive apps also increase, the need arises for a way to analyze the graphical capabilities of the devices on the market in order to continue pushing the limits and producing the next generation of games and apps. The purpose of this thesis paper is to provide an understanding of the graphical capabilities of current Android OS based mobile devices through performance benchmarking using a customizable benchmarking solution. An Android application called “MOBILESpeed” will be developed to be used for benchmarking using scene decomposition which will allow for better analysis and identification of performance bottlenecks. This application will serve to analyze a device’s graphical performance through predefined and customizable benchmark test scenes.

This thesis will describe the concept of benchmarking, provide information on mobile graphics hardware and architecture, explain the fundamentals and features of OpenGL ES, and also present in detail the tools and methods used to produce the
MOBILESpeed benchmarking application from design to implementation. Lastly, a range of devices will be benchmarked and analyzed using MOBILESpeed. The purpose of MOBILESpeed is to provide objective graphical performance data for Android devices so that those looking to acquire a device for graphically intensive applications such as gaming can make a more informed choice.
INTRODUCTION

In recent years, the rapid advancement of mobile device hardware and software has led to both an increase in popularity of mobile-based video games, and to the development of more graphically intensive applications. Due to the rapid advancement in technology and the market demand, big name mobile device manufacturers like Samsung, HTC and LG are frequently releasing new products or newer versions of existing products with improved hardware specifications. With each cycle of new device releases, gamers, mobile device reviewers, and others rely on up-to-date benchmarking applications to gather performance metrics, however the rate at which hardware is being released can become problematic for them if the benchmarking software is not updated.

The current existing benchmark applications can be classified into two types: General hardware performance benchmarks, and graphics performance benchmarks. As far as this paper is concerned, the focus will be on benchmarks specifically designed for graphics, however it is important to keep in mind that graphical performance strongly relies on the capabilities of the system as a whole, and not just the graphics processing hardware.

As one can assume, better performing devices may require modifications to existing benchmark software in order to adequately stress the graphical capabilities of the new hardware in order to provide the most useful results. The solution to this problem is to create a flexible, customizable benchmark that allows a user to control both the intensity level of the tests and scene configuration of the benchmark. Existing benchmark applications are limited by their lack of user-customizability. By offering a level of control and customization, the lifespan and relevance of a benchmarking application will be prolonged.
Project Goal

The inspiration to develop a more customizable benchmarking tool in order to assess the graphical performance of mobile devices came from the lack of availability of a tool that provides additional levels of control to users than a standard benchmark, and also decomposes a scene to expose more of the internal implementation details of the graphics on screen. These things naturally assume that the user has some background knowledge of graphics as this benchmarking tool is targeted towards such a user. MOBILESpeed can be used by developers to aid in identifying bottlenecks and assessing system capabilities. MOBILESpeed does not have a scoring system like some other benchmarks do, instead provides results in terms of detailed frame rate measurements.

This paper will introduce the reader to graphics processing on mobile Android devices, help understand the importance of graphics benchmarking, and provide design and implementation details for creating a customizable benchmarking application. Once the benchmark has been developed, various devices will have their performance measured using MOBILESpeed and the results will be presented and analyzed to demonstrate the benefits of this benchmarking application.

Approach

The technical approach to this project is overviewed in this section, with complete implementation details provided throughout the paper.
A set of software and hardware resources were utilized for the development, testing and analysis of the MOBILESpeed benchmarking application. The core software resources used include:

- Java programming language
- OpenGL ES 2.0 library
- Eclipse Juno with ADT plugin (minimum version 3.7.2 is required)
- Java JDK 6 / Java Runtime Environment 1.8.0
- Git version control system

The Java programming language is used for this project because it is the preferred development language for Android. Although Android development environments and compilers for languages like C++, C# and Python exist, Java was chosen because it provides for cross-platform support, ease of development, security, and stability. The OpenGL ES graphics library is the most common way to render graphics on embedded mobile devices and is accompanied by ample documentation and resources. All graphics
in the application are constructed from scratch, without the aid of modeling software, using purely OpenGL library routines. Lastly, Git was chosen for version control management during development due to its popularity, reliability, and no cost code repositories. The hardware resources used for development include:

- Windows PC
- Acer Android tablet, Samsung Android tablet.

Development requires any computer capable of running the Eclipse IDE with the Android Development Tools plugin. For this application, development occurred on a Windows 7 based PC. Testing and debugging of the application was performed on two different Android tablets connected to the development PC through a USB port. The Acer tablet supported plug and play and required no user configuration to work with Eclipse when connected. The Samsung tablet required installation of Samsung’s “Kies” software which provides the drivers needed for the tablet to be recognized when connected (all Samsung tablets require Kies installation). Also, on all devices, USB debugging must be enabled through the Settings menu (Settings → Applications → Development → USB Debugging). The items mentioned here complete the required hardware setup.

**Android and Mobile Graphics**

The Android platform was chosen for this study due to the significant variation and availability of devices when compared to Apple’s iOS platform. The Android market is quite diverse when it comes to both software and hardware. This diversity is part of the motivation for this study, as the drawback of having a wide selection is being able to decide on the device best suited for your needs if your interest is something like gaming. The Android support team continuously tracks information about devices in use and
provides this information to developers on their site. The tables below represent

<table>
<thead>
<tr>
<th>Version</th>
<th>Codename</th>
<th>API</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>Froyo</td>
<td>8</td>
<td>0.4%</td>
</tr>
<tr>
<td>2.3.3 - 2.3.7</td>
<td>Gingerbread</td>
<td>10</td>
<td>6.9%</td>
</tr>
<tr>
<td>4.0.3 - 4.0.4</td>
<td>Ice Cream Sandwich</td>
<td>15</td>
<td>5.9%</td>
</tr>
<tr>
<td>4.1.x</td>
<td>Jelly Bean</td>
<td>16</td>
<td>17.3%</td>
</tr>
<tr>
<td>4.2.x</td>
<td></td>
<td>17</td>
<td>19.4%</td>
</tr>
<tr>
<td>4.3</td>
<td></td>
<td>18</td>
<td>5.9%</td>
</tr>
<tr>
<td>4.4</td>
<td>KitKat</td>
<td>19</td>
<td>40.9%</td>
</tr>
<tr>
<td>5.0</td>
<td>Lollipop</td>
<td>21</td>
<td>3.3%</td>
</tr>
</tbody>
</table>

Table 1: Android OS Platform Versions as of March 2, 2015

<table>
<thead>
<tr>
<th>OpenGL ES Version</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>67.5%</td>
</tr>
<tr>
<td>3.0</td>
<td>32.5%</td>
</tr>
</tbody>
</table>

Table 2: OpenGL ES Versions Used on Android as of March 2, 2015

Based on this distribution data, the decision was made to target platform version 4.4 and
OpenGL ES version 2.0 for the benchmarking application as these are the most widely
used versions and reduce the likelihood of compatibility issues.

**Android Graphics Architecture**

Graphics rendering APIs are provided by the Android framework to support both
2D and 3D graphics. The 2D graphics, which include widgets such as menus, status bars,
and buttons, are drawn using the Canvas API which is responsible for drawing all
standard and user created View objects to the screen. A View is an area on the screen that
contains user interface components such as buttons and text fields. To efficiently draw
Views, Canvas API calls are converted to OpenGL by Android which then allows the system to perform hardware accelerated drawing. Hardware acceleration is a feature that makes use of the GPU hardware, rather than the CPU to increase graphical performance. For 3D graphics, the OpenGL ES API is used directly to perform the drawing.

Both Canvas and OpenGL ES APIs draw everything to a graphics component called a surface. The surface’s role is to write information regarding graphical data to a buffer queue which is then processed by other lower-level components whose responsibility is to read that data and compose what is viewed on the display.

![Android’s Surface Rendering Pipeline](image)

Figure 2: Android’s Surface Rendering Pipeline

The complete surface rendering process contains both high-level components (which the developer directly interacts with) and low-level components (which are controlled by the system) and works in the following way:
1. Graphics data is created from numerous sources and written to graphics buffers. These are the Producers depicted in the surface rendering pipeline diagram above. An example of a Producer is a video game written in OpenGL ES, or a custom view created with Canvas. It should be noted that all graphics buffers are implemented as buffer queues and can operate in modes that allow or prevent buffer overrides for data management.

2. The graphics buffers are read and processed using Window Manager data. These are the Consumers depicted in the surface rendering pipeline diagram above. The Android system uses a service called SurfaceFlinger which reads the numerous graphics buffers, obtains information about the visible windows, and begins the composition process by working with the Hardware Composer component.

3. Composition of the buffers is performed to create a displayable output. The Hardware Composer (HWC) is the hardware abstraction layer and works with SurfaceFlinger to perform the final composition. The HWC is responsible for determining the most optimal way to perform buffer composition with the hardware being used. Each Android device manufacturer will have its own specific implementation of the Hardware Composer.
Further details regarding the SurfaceFlinger’s and Hardware Composer’s roles and functions within the Android graphics architecture are beyond the scope of this paper. As mentioned earlier, OpenGL ES renders to a graphics component known as a “surface” which is simply a dedicated piece of memory that can be composited into the view system. MOBILESpeed utilizes the GLSurfaceView implementation of this component. A GLSurfaceView provides an area on the screen to display graphics rendered with OpenGL. Although this GLSurfaceView fills the entire screen, it still technically resides behind the user interface surface and is processed on a separate rendering thread to avoid interference with the UI surface processing and provide optimal performance. In relation to the Android graphics processing scheme, the GLSurfaceView will serve as a Producer, writing data to the buffer queue which then the SurfaceFlinger and Hardware Composer components will later composite the graphics from, along with the UI surface and any other existing view layer.

Understanding the internal details of how the Android system processes graphics is an important step towards understanding how to design a benchmarking application to
provide meaningful and accurate performance metrics. The next step is to become familiar with the OpenGL ES API in order to effectively use the features of the library.

**OpenGL ES 2.0 Overview**

OpenGL is a free, portable, and cross-language desktop graphics API for drawing 2D and 3D computer graphics and is the most widely used open graphics standard today. The API allows for software applications to utilize the graphics processing unit (GPU) on a computer system to render high resolution, high performance graphics. OpenGL is implemented in hardware by graphics vendors, making it very efficient, but it can also be made available through a pure software implementation.

OpenGL ES is a variation of the original desktop OpenGL library and was specifically created for use with embedded systems. The vast functionality provided by the original OpenGL API was reduced to a smaller, more essential set of functions to be used with embedded devices. The initial version of OpenGL ES (version 1.0) was introduced in 2003 and has since gone through several revisions (current version 3.1), each adding more and more functionality to provide additional capabilities for creating high performance graphics. As the versions progressed, more of the fixed-pipeline functions of version 1.0 were eliminated in favor of shaders, which offer more control and flexibility to the graphics developer. OpenGL ES 3.1, 3.0 are backwards compatible with version 2.0, however 2.0 is not backwards compatible with any prior version. Currently, the majority of active Android devices support version 2.0 which is why it was decided to develop MOBILESpeed using this version of the API.
<table>
<thead>
<tr>
<th>OpenGL ES API Version</th>
<th>Supporting Android Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 - 1.1</td>
<td>1.0 and higher</td>
</tr>
<tr>
<td>2.0</td>
<td>2.2 (API 8) and higher</td>
</tr>
<tr>
<td>3.0</td>
<td>4.3 (API 18) and higher</td>
</tr>
<tr>
<td>3.1</td>
<td>5.0 (API 21) and higher</td>
</tr>
</tbody>
</table>

Table 3: OpenGL ES Versions Supported by Android

In order to provide an understanding of the features of OpenGL and how it processes graphics, a high-level overview of OpenGL ES 2.0 architecture and features is presented in this section beginning with the processing pipeline. Keep in mind that all mentions of “OpenGL” for the remainder of this section are specifically referencing OpenGL ES 2.0 and not desktop OpenGL.

**Graphics Processing Pipeline**

The OpenGL ES 2.0 pipeline is quite different from the fixed-function pipeline architecture of version 1.0/1.1. In fact, it is so different that code written for one architecture is incompatible with the other. The programmable aspect of the pipeline means that a developer can write code to perform custom operations on vertices and fragments as they are processed, rather than relying on OpenGL to do it. The custom developer code that is executed is referred to as a “shader program” or “shader” for short and written in OpenGL ES Shading Language. In version 2.0 of the API, the two types of shaders available are vertex shaders, for operating on vertices, and fragment shaders, for operating on fragments. The thing to note here is that the programmable pipeline’s benefits come at a cost of increased code and processing complexity.
The processing pipeline can be most simply described as the process needed for taking a mathematical expression of an object to be drawn, transforming it to a grid of colored pixels representing that object, and displaying it on a screen. As one can imagine this process is quite complex. When OpenGL is requested to draw an object, it must take the data and go through a sequence of processing steps before the graphics can be displayed on the screen; this is known as the rendering pipeline. The rendering pipeline can vary among OpenGL versions because of the operations or functionality one version supports over another. Higher versions of OpenGL provide more programming flexibility and control through additional programmable pipeline stages, but do not necessarily have to be utilized. The term “programmable” means that a stage in the pipeline can execute a custom written program that performs operations on the incoming data. In the OpenGL ES 2.0 pipeline there are two programmable stages: the vertex shader, and fragment shader. The remaining stages are not programmable. The processing diagram shown above details all the stages of the pipeline. The pipeline execution begins with an API call.
that submits a set of vertices for processing. The GPU receives the vertex data and begins processing each vertex in the stream by first passing it through the vertex shader. The vertex shader takes an input vertex, applies some programmer specified processing on it (e.g. a transformation) and produces an output vertex for the next stage. The outputted vertices from the vertex shader then go into the primitive assembly stage where they are processed and converted into points, lines, or triangles, the basic shapes from which all complex shapes are made up. Once the primitives are constructed, they proceed to the rasterization stage which is responsible for mapping each point of the primitive to a grid square (pixel) in the screen’s window coordinate system. After the primitive is associated with a grid square, the square is assigned a color and depth (used to distinguish overlapping objects) value and the output from this stage is referred to as a “fragment” or potential pixel. The fragment shader then takes the resulting output data from the rasterization process and like the vertex shader applies some additional programmer specified processing on it (e.g. texturing). Finally, each outputted fragment from the fragment shader undergoes a series of special tests to determine its eligibility to be added to the framebuffer. If the fragment passes all required tests, it is written to the framebuffer which contains the data needed to draw the next frame of graphics on the screen. This concludes the OpenGL processing pipeline.

Features

The OpenGL ES 2.0 API introduced a vast set of capabilities to the graphics programmer, all of which provide improved rendering performance and aid in creating complex effects that may have been otherwise very difficult. A summary of the notable features of version 2.0 include:
- Programmable graphics processing pipeline
- Vertex and Fragment shader programming (using GLSL ES)
- No backwards compatibility with version 1.0/1.1
- Allows graphics vendors to customize extensions

**Mobile GPU Architecture**

Before attempting to develop a benchmark, another important aspect of the system to consider and understand is the GPU architecture. Without some low-level knowledge of the GPU and driver implementation, mistakes in the benchmark design can be made leading to inaccurate performance measurements.

The majority of mobile devices have graphics processing units integrated into the system’s SoC (System on a Chip) processor. Mobile GPUs differ architecturally from traditional desktop GPUs in terms of their rendering schemes in order to achieve the best possible performance under more restrictions. The two main graphics processing architectures are Immediate Mode Renders (IMR) and Tile-Based Renderers (TBR). IMRs are common in desktop GPUs like Nvidia’s GeForce series and AMD Radeon series, and are less complex implementation-wise than TBRs.

![Immediate Mode Rendering Diagram](image)

**Figure 5: Immediate Mode Rendering Diagram**

Immediate Mode Rendering architectures begin processing and drawing frames as soon as they receive the commands, hence the “immediate” in the name. A full scene is drawn
in only one pass. This technique could be considered brute force since it doesn’t take into account every object that needs to be drawn on the screen prior to initiating the drawing process in order to eliminate unnecessary work. For example, in a typical scene that contains many polygons that partially overlap one another, an IMR will perform all the necessary computations for every polygon, even the ones that are obstructed and won’t be visible in the final frame. This event is referred to as overdraw and is very wasteful of valuable processing resources. However, GPUs that utilize IMR are usually designed with the intention of being in a desktop system where resources like memory bandwidth and power consumption are not a major concern.

Tile-Based Renderers attempt to optimize the drawing process by modifying some steps in the processing pipeline and adding others with the intention of reducing wasteful processing in an effort to speed up drawing, reduce bus bandwidth, and conserve power, all of which are valuable in a mobile system. A common TBR rendering implementation is Deferred Rendering, which aims to delay fragment processing until necessary.

![Tile Based Deferred Rendering Diagram](image)

**Figure 6: Tile Based Deferred Rendering Diagram**

The basic concept behind Tile-Based Deferred Rendering is to take a frame to be rendered and divide it into pieces, creating a grid of small tiles where each tile can be thought of as an N x N area of pixels. Once tiling is complete, the GPU has references to associate each piece of geometry in the scene with a tile and can begin rasterization. The
GPU then determines which parts of the geometry are visible to the user through a process called hidden surface removal (this is done on a per-tile basis) before shading and applying textures to only what passes the visibility test. In Tile-Based architectures, all rendering work is performed tile-by-tile which is why multiple passes are required for a single frame to be drawn. The architecture also frequently reorders the draw commands to allow the GPU to execute them in the most efficient manner, unlike IMRs which execute them in the order received.

There are various implementations of Tile-Based rendering. Deferred Rendering is just one form of it most specific to PowerVR GPUs, which are one of the most popular. Other common mobile GPU architectures are closer to Immediate Mode Renderers with the addition of scene tiling or early pixel rejection into the processing pipeline.

The advantages of Tile-Based Rendering:

- Lowers power consumption by eliminating redundant GPU processing.
- Reduces memory bandwidth since tiling allows the GPU to work with a small set of data at a time. All the data for a tile can be loaded into the fast but small on-chip GPU memory, minimizing the frequency of external memory access.
● Inherently parallel design (simple to parallelize processing).

The disadvantages of Tile-Based Rendering:

● More complex implementation.

● Does not work well for scenes with complex geometry and high polygon counts.
  ○ Too much overhead to pre-process geometry.
  ○ Triangles overlapping multiple tiles are processed multiple times.

Tile-Based Renderers and the various modified Immediate Mode Renderers are very popular in mobile graphics architectures for their ability to deliver a balance of performance and efficiency. The processing architecture of the target GPU is an important thing to consider when developing a benchmark.

**Trade Study**

Research was conducted to discover graphics benchmarking applications currently available for the Android platform with two intentions: studying the features provided and for comparison to MOBILESpeed. The list of applications discovered was narrowed down to two popular benchmarking apps: GFXBench 3.0 and 3DMark.

GFXBench is developed by Kishonti Informatics and is available at no cost through the Google Play store. It provides a variety of benchmarks focused on testing graphics performance, system rendering quality, stability, and power consumption. It also provides an information page which details the hardware specs and API versions of the device. Version 3.0 of this app features a variety of OpenGL ES 2.0 benchmark tests. The set of tests available in this app are categorized into high, low, and special tests as follows (Note: the description of each benchmark test is provided by the product vendor):
- **T-Rex:**
  Highly detailed textures, materials, complex geometry, particles with animation textures and a post-process effect: motion blur. The graphics pipeline also features effects: planar reflection, specular highlights and soft-shadows.

- **ALU:**
  Measures the pure shader compute performance of your device by using a complex fragment shader and rendering a single full-screen quad.

- **Alpha Blending:**
  Measures the device’s alpha blending performance by rendering semi-transparent screen-aligned quads with high-resolution uncompressed textures.

- **Driver Overhead:**
  Measures the OpenGL driver’s CPU overhead by rendering a large number of simple objects one-by-one, changing device state for each item. The frequency of the state changes reflects real-world applications.

- **Fill:**
  Measures the device’s texture performance by rendering four layers of compressed textures. This is a common scenario in gaming.

- **Battery Test:**
  This combined test measures your device’s battery life and FPS stability in a gaming scene by running the same high-level test for 30 times.
- Render Quality:

  Measures the visual fidelity of your device in a high-end gaming scene. The score is the peak signal-to-noise ratio (PSNR) based on mean square error (MSE) compared to a pre-rendered reference image.

  Each test gives the user an option to render the test on-screen or off-screen, with the benefit of the off-screen mode being that the scene is rendered in 1080p resolution. The results produced by GFXBench are presented in terms of total number of frames rendered and the frames per second. Additionally, the app allows the user to view and compare their device’s test results with an extensive database of other devices.

  3DMark is another graphics-centric benchmarking app from Futuremark. The aim of 3DMark is to test both CPU and GPU performance using a single scene called “Ice Storm” which advertises to be highly demanding on even the most capable devices. The Ice Storm test measures CPU performance with complex physics and GPU performance with processing a high vertex count followed by a high pixel count. It is available in a normal and extreme intensity level, with the normal intensity version featuring an OpenGL ES 2.0 implementation rendered in 720p using low resolution textures, while the extreme renders in 1080p using high resolution textures. There are three test modes the Ice Storm scene runs in:

  - Graphics Test 1:

    Stresses the hardware’s ability to process lots of vertices (530,000 - 580,000 vertices/1.9 - 4.4 million pixels) while keeping the pixel load relatively light. Hardware on this level may have a dedicated capacity for separate vertex and pixel processing. Stressing both capacities individually reveals the hardware’s
limitations in both aspects. Pixel load is kept low by excluding expensive post processing steps, and by not rendering particle effects.

- **Graphics Test 2:**
  Stresses the hardware’s ability to process lots of pixels (79,000 - 89,000 vertices/7.8 - 18.6 million pixels). It tests the ability to read textures, do per pixel computations and write to render targets. The additional pixel processing compared to Graphics test 1 comes from including particles and post processing effects such as bloom, streaks and motion blur. The numbers of vertices and triangles are considerably lower than in Graphics Test 1 because shadows are not drawn and the processed geometry has a lower number of polygons.

- **Physics Test:**
  Benchmarks the hardware’s ability to do gameplay physics simulations on CPU.
  The GPU load is kept as low as possible to ensure that only the CPU’s capabilities are stressed.

Ice Storm also provides real-time FPS and total frames rendered counts on-screen to show the frame rate fluctuations based on the dynamics of the scene. The final result produced by the benchmark is presented as an FPS value per test and an overall average score which factors both graphics and physics performance values.

Overall, both benchmark apps offer a solid suite of tests but lack to offer a customizability aspect to benchmarking. The MOBILESpeed benchmark app aims to provide a similar approach to benchmarking with a collection of individual tests that focus on measuring performance of specific areas while offering the user more control and customizability when it comes to the intensity level of the tests.
BENCHMARKING CONCEPTS

Benchmarking (or a Benchmark) in terms of computing or a computer system by definition is “a standardized problem or test that serves as a basis for evaluation or comparison.” Its purpose is to serve as a reference metric to which other measurements can be made and compared against.

The Purpose of Benchmarking

It is important to understand that benchmarking provides knowledge of a system’s potential and its maximum capabilities. Without actually running a benchmark test, it is often difficult to subjectively determine the more favorable performance outcome between two systems with similar hardware specifications because of the number of factors involved. The complex nature of graphics processing is highly susceptible to performance issues caused by both software and hardware.

The purpose of a benchmark test is to identify bottlenecks in the processing chain. Once the bottleneck is identified, analysis will determine if and how it can be removed or mitigated. Benchmarking a system using a graphics application is ideal because of the processing power required by something like a video game. Displaying visually impressive graphics is very demanding on a system and truly exercises and stresses all components of the system (e.g. RAM, hard disk, CPU, GPU), which is why it makes for an optimal benchmark.

Frame Rate

Frame rate (or frame frequency) is the speed at which an image can be refreshed or redrawn. Collecting frame rate information is the most popular benchmark metric for measuring performance in video games, and it can also be used for other graphically
intensive applications. High and stable frame rates produce fluid animations and provide a smoother, more interactive user experience, however, lower and more stable frame rates are preferred over higher and unstable ones as instability will result in stuttering and choppiness of the images displayed on the screen. The two ways to look at frame rate are:

**Frames Per Second (FPS)**
- The number of unique consecutive images per second (called “frames”).

**Frame Time**
- The frame latency, or time between two frames. This is in inverse of Frames Per Second.

To understand the usage difference between FPS and frame time, keep in mind that FPS is a measurement of performance; however frame time is a more effective way to measure performance improvement. This is due to the linear nature of frame time values.

Consider a video game that runs at a frequency of 500 FPS on a particular system, meaning the frame time will be 2 milliseconds. As more elements are added to the game, the computational complexity increases causing the FPS rate to decrease in half to 250 FPS. Now a difference of 250 frames seems quite significant but there has only been 2 additional milliseconds of processing time added, which is not very much. This is why frame time should be used for performance analysis reasons.
Most Android device drivers limit the maximum frame rate performance of an application to 60 frames per second (16.67 milliseconds frame time) by enforcing vertical synchronization (V-Sync). There is no elegant way to bypass this limitation so MOBILESpeed’s frame rate output will, on most devices, be no higher than 60 FPS.

**Bottlenecks**

Understanding how to measure frame rate of an application is only half the performance optimization effort. The other, more challenging half is understanding why the application’s graphics may not be drawing at the most optimal rate. In computing, the term “bottleneck” is commonly used to describe a situation where the performance of a process is limited by some component, resulting in lower throughput. Be aware that the source of a bottleneck can stem from either hardware or software. If a function in the software implementation is the bottleneck, traditional techniques to uncover the source include code analysis and the use of performance profilers. Examples of potential software bottlenecks in an OpenGL application include using deprecated APIs, inefficient code logic that limits data flow to the GPU, complex vertex and fragment shader calculations, and even poor implementation of functionality in the graphics driver.
If a component in the hardware configuration is the bottleneck, it may potentially be upgradeable to help improve performance. Examples of upgradable components include CPU, GPU, and RAM. However, on a mobile platform, hardware bottlenecks that software can’t mitigate are more difficult to overcome because mobile phones don’t have the flexibility like desktop systems to have their components modified or upgraded. Each component’s performance capacity contributes to the overall graphics performance of the system, meaning if a device has a high-end GPU but poor RAM and CPU, the GPU will not be able to perform to its capabilities because the other components degrade, or bottleneck the system as a whole.

**V-Sync**

Vertical Synchronization (V-Sync) is an option to synchronize the frame rate of an application with the refresh rate of the monitor, or more technically, the process of aligning the back and front framebuffer swapping interval with the hardware’s frequency of updating what is presented on the display. The framebuffer is what contains the data that represents the next image (or frame) to be displayed on the screen. In what is known as double-buffered system, there exists a front buffer, which contains the data that composes the frame currently visible on the screen, and a back buffer, which is an off-screen buffer that the GPU uses to compose the next frame to be display on the screen. The swapping of these buffers means to move the contents of the back buffer to the front buffer. This action is performed immediately once the drawing of the frame is complete if V-Sync is not enabled. With V-Sync, the buffer swap only occurs when the screen is about to be refreshed by a hardware controlled event called “v-blank.” Consider a standard monitor that refreshes its screen at a rate of 60 Hz (every 16.67 milliseconds)
displaying an application that requires 10 milliseconds for the GPU to render a single frame for it. V-Sync will introduce a 6.67 millisecond stall before performing the buffer swap to ensure that the swap is in line with the v-blank signal. Once the v-blank occurs, the front and back buffers are swapped, the back buffer is free again for the next frame to be rendered into it, and the process repeats. To summarize this process:

1. GPU draws to back framebuffer. Front framebuffer is being display on screen.
2. GPU completes drawing to back framebuffer.
   a. V-Sync Disabled: GPU swaps back/front buffer contents immediately.
   b. V-Sync Enabled: GPU waits for signal to swap back/front buffer contents.
3. Screen displays the new frame. GPU begins to draw next frame to back buffer.

Now, the question to be asked is why is V-Sync needed when it seems to add an unnecessary delay in the rendering resulting in a lower frame rate? The answer is that it helps eliminate two undesirable visual artifacts that are commonly seen: stuttering and screen tearing. Stuttering is caused by significant dips and rises in the time it takes to render successive frames. With V-Sync, the frames are forced to be displayed at a more regular interval (masking the variation in rendering time per frame) to the user thus eliminating the stuttering effect.

The other undesirable effect that can be seen without V-Sync is screen tearing. Screen tearing is observed when the screen displays two different frames concurrently. This occurs most often when the GPU is rendering frames and trying to display them significantly faster than the monitor is able to display, causing half of the current frame and half of the next frame to occupy the screen during one display cycle. Another benefit of V-Sync is that it reduces the number of unnecessary frames that are drawn by the
GPU. Drawing frames faster than the screen can display them results in discarded frames and additional power consumption which is detrimental to a device running off battery power. For benchmarking purposes in general, it is recommended to disable V-Sync (if possible) since the intent is to measure maximum performance without concern for visual aesthetics. However, as mentioned earlier in the section, most Android devices don’t provide an option to disable V-Sync.

**Antialiasing**

Antialiasing (AA) is an algorithm used in computer graphics to make jagged lines appear smoother so the image quality is perceived to be higher, particularly on lower resolution displays. It is another important concept to be aware of because of its popularity and potentially significant performance impacts to benchmarking.

A screen is made up of a rectangular grid of squares called pixels that are used to try to accurately draw an object. Most objects naturally have some degree of roundness to them which makes it impossible to represent a perfectly smooth diagonal line or curvature of a circle on a screen.

![Figure 9: Example of Antialiased vs. Aliased Edges](image)

In order to soften the appearance of a rough edge, the antialiasing algorithm attempts to fill in the pixels around the edges of the shape with a color that is near the color of the edge itself. In the image of the diagonal line seen above, notice the transition from the
black color of the line to the white empty space is softened by a gray-toned pixel, which resulted from the blending of the two colors. Once antialiasing is applied, the stairstep look to the edge is not as apparent anymore. If this same image was displayed on a screen with a very high resolution, antialiasing would not be needed because the size of the individual pixels would make the edges unnoticeable.

The downside to antialiasing is that it adds additional processing to every frame, thus increasing the time required to draw a frame. The two main factors that determine how much of impact applying antialiasing adds to drawing time is the complexity of the algorithm and the number of samples it takes. MOBILESpeed provides an option to enable one of the most common types of antialiasing known as multisampling.

Now that a few existing benchmark applications along with the concepts of benchmarking have been introduced, the next sections will discuss some of the functions and features that the MOBILESpeed benchmarking application will provide, followed by design and implementation details.
APPLICATION REQUIREMENTS

Following a period of research and analysis of existing benchmark applications on Android and various other platforms, a list of commonly provided features and options was compiled and used to aid in writing MOBILESpeed’s requirements. This section details the functionality required for the MOBILESpeed mobile graphics benchmarking application. The requirements are categorized as functional, user interface, and performance requirements.

**Functional**

[1.0] The application shall provide a predefined set of benchmark tests that can be run through a single button click from the app’s home screen.

[1.1] The application shall provide a list of all the tests available to the user for customization of a benchmark scene.

[1.2] The user shall be able to interrupt a currently running benchmark test at any given time and return to the home screen or test selection screen.

[1.3] The user shall be able to view the results from the last benchmark test run.

[1.4] The user shall be provided hardware details about their device.

[1.5] The application shall support a minimum of Android SDK version 14 and a target SDK version of 19.

[1.6] The application shall only be available for devices supporting OpenGL ES 2.0 and above. All benchmark tests will be written using OpenGL ES 2.0 features.

**User Interface**

[2.0] The user shall be able to enable or disable each individual benchmark test through a checkable interface, allowing any combination of tests to be selected.
[2.1] The user shall be able to manipulate the quantity of objects drawn in each benchmark test when applicable through a number picker.

[2.2] The application shall provide navigation among the various screens through the use of navigation tab or navigation drawer panels.

[2.3] All lists and tables containing objects shall be scrollable when the list or table extends past the viewable area.

[2.4] Version, licensing and support contact information for the application shall be provided to the user.

**Performance**

[3.0] The application shall use Android’s large heap memory model setting in order to allow for large quantities of geometry to be loaded into memory.

[3.1] The application shall check for OpenGL ES 2.0 compatibility on the target device.

[3.2] The application shall place an upper limit on the quantity of geometric objects the user can request to be drawn for each benchmark test.

[3.3] The application shall provide feedback response for all buttons.

[3.4] Each individual benchmark test shall take a maximum of 60 seconds to run.

[3.5] The application shall prevent the screen from dimming or locking.

Incorporating the essential requirements listed in this section, along with any others discovered during design and implementation will ensure that a robust, complete, and high-quality benchmarking application is developed.
APPLICATION DESIGN

Features

The application design of MOBILESpeed is driven heavily by its customizable benchmarking features. As mentioned earlier, the goal of MOBILESpeed is to provide controls for benchmark test customization to the user, rather than simply providing just a pre-defined set of tests which is what is usually done by traditional benchmarking software suites. Benchmark test customization is the highlighted feature of MOBILESpeed, however it does also provide a variety of other key features including:

- **User-friendly interface:** the intuitive UI design allows a user with little knowledge of the app to create and run a benchmark test.

- **Pre-defined benchmark tests:** a user can bypass the benchmark customization process and simply run a pre-defined test scenario where each individual benchmark is run with increasing levels of stress.

- **Customizable benchmark test creation:** allows a user to create a custom scene to be rendered and benchmarked.

- **Device hardware and capability information:** provides the user hardware specs for their device as well as SDK/API and operating system details.

- **Accurate performance metrics:** frame time and frames per second (FPS) results are provided with high accuracy.

MOBILESpeed is designed to be a completely functional Android app with the potential for release on the app market. It is compatible with many device models and Android OS versions.
Software Architecture

This section presents a comprehensive look at the software’s architecture through detailed descriptions and various architectural views of the application, including use case, package, class and sequence. One of the main goals of this architectural discussion is to explain the design choices that were made for the application.

Architectural Goals and Constraints

The primary design goal for this software system is to isolate the user interface, data, and OpenGL components from one another in an effort to reduce dependencies and also follow a Model-View-Controller design pattern. This unfortunately was not intuitive to do. It is tough in Android to implement true MVC for several reasons, the main being that Android heavily favors interface implementation and inheritance relationships over composition, which makes it rather difficult to decouple controller, view and data classes.

The software system does not have many technical constraints imposed upon it, with the exception of Java as the programming language, XML for UI layout creation, Android SDK minimum version 14, and OpenGL ES version 2.0.

UML Diagrams

An insight into the application’s architecture and detailed design is best presented through UML diagram illustrations. This section will provide a look at the software design from a high level perspective with use cases, package organization diagrams, and major sequence diagrams. The finer details of the package contents, which include detailed class diagrams is available in Appendix A.
Use Cases

Figure 10: Use Case for Main Interaction

The use case in the figure above depicts the primary types of interaction the user can perform on the application from what is considered the home screen of the application (this is the screen that is presented upon application startup). All interactions that end with the word “screen” indicate that a tab-styled button is available to navigate to the indicated screen. The option “Run all tests” starts the benchmark with a preconfigured set of tests that run in sequence of increasing levels of stress. The option “Create custom test” navigates the user to the Tests screen which allows selection and configuration of the available benchmarks (see figure below).
Figure 11: Use Case for Test Customization

The use case in the figure above depicts the interaction available to the user in the Tests screen of the application. The Tests screen provides interaction through a checkable interface for adding tests to the benchmark, along with spinners for configuring the quantity of objects to render in the scene.
The application’s classes are assigned into packages based on their logical grouping and functionality. The available packages include:

- **opengl**: contains the OpenGL classes that perform the rendering of scenes.
• **ui**: contains the Activity and Fragment classes used by Android to create the user interfaces.
• **model**: contains classes that implement the data and logic of the UI components. This is done to separate logic and user interface code.

• **adapter**: contains classes that act as a bridge between a view class and data class associated for that view.
Figure 16: Package Contents - adapter

- **util**: contains utility classes used by all packages.

Figure 17: Package Contents - util

- **sandbox**: this is where experimental classes reside. Once a class is ready for use, it is moved out of this package. This package is not used in the final product therefore no additional documentation will be provided from this point on.

Classes

This section provides a listing and description of the classes in the application to help understand MOBILESpeed’s design. For complete detailed class diagrams which
include attributes and methods, along with high-level associations and dependencies among classes within a package, refer to Appendix A.

UI - Class Details

- **MainActivity**: serves as the entry point into the application. Upon app startup, the operating system initiates a call to this class to perform initial resource initializations and to render the views that are presented to the user. MainActivity creates the Android fragments that represent each tab in the navigation bar.

- **HomeFragment**: generates the view content and handles input for the Home tab.

- **TestsFragment**: generates the view content and handles input for the Tests tab.

- **ResultsFragment**: generates the view content and handles input for the Results tab.

- **DeviceInfoFragment**: generates the view content and handles input for the DeviceInfo tab.

- **AboutFragment**: generates the view content and handles input for the About tab.

- **SplashScreen**: an Activity class that generates the splash screen containing the MOBILESpeed logo seen during a cold start of the application.

Model - Class Details

- **Tests**: container to represent which benchmark tests are selected and the quantity value for each test as inputted by the user. This class is used to communicate between the 2 main Activity classes (MainActivity, GraphicsActivity) and contains methods to support serialization and deserialization of the object.

- **DeviceInfo**: responsible for obtaining system and device hardware information and serving as the model class for the DeviceInfoFragment UI class.
- **Metric**: represents a single row of data displayed in the scrollable list on the Results screen. Each object of type Metric contains a title (the name of the test run) and value (the result of the test).

**Adapter - Class Details**

- **TabsPagerAdapter**: extension of Android’s FragmentPagerAdapter. Responsible for determining the correct Fragment class to display based on the selected tab in the Action Bar.

- **ResultsArrayAdapter**: extension of Android’s ArrayAdapter. Responsible for inflating the view that represents each row in the scrollable list on the Results screen.

**Util - Class Details**

- **MyApp**: extension of Android’s Application class. Its primary purpose is to provide a handle to the application context to the various Activity and Fragment classes. The application context is required to access certain app resources.

- **Stopwatch**: timer class used to measure code execution time. Stopwatch utilizes Java’s System class clock utility functions.

- **MultisampleConfigChooser**: used to create an OpenGL ES rendering context that supports multisampling (for antialiasing).

- **MathUtils**: contains general math utility functions.

- **CheckableLinearLayout**: custom layout class to create the selectable boxes used in the Tests screen of the app.

- **Consts**: container for globally used constant value definitions.
- **PageTransformerZoomOut**: used to add a zooming in/out animation effect when navigating through the ActionBar tabs.

- **PageTransformerDepth**: used to add a depth animation effect when navigating through the ActionBar tabs.

### OpenGL - Class Details

- **GraphicsActivity**: contains the Activity lifecycle methods for the opengl component of the application.

- **SceneGlSurfaceView**: view class that hosts the opengl renderer.

- **SceneRenderer**: creates and executes the opengl benchmark tests.

- **GLSLShader**: provides functions to read, compile, and link shader files.

- **GLDrawUtils**: collection of geometry drawing functions.

- **GLVertex3D**: used to represent a 3D point.

- **GLPosition3D**: used to represent a 3D location in space.

- **GLColor**: used to represent color for a point in RGBA format.

- **GLTest**: interface class implemented by all benchmark test classes.

- **GLPointTest**: renders a scene of points.

- **GLLineTest**: renders a scene of lines.

- **GLTriangleTest**: renders a scene of triangles.

- **GLCollisionTest**: renders a scene of colliding balls. Contains nested class Circle.

- **GLTextureTest**: renders a scene of textures.

- **GLLightTest**: renders a scene with lighting effects.

- **GLStaticTest**: renders a scene with a single frame multiple times.
- **GLBusBandwidthTest**: renders a blank scene that repeatedly performs a large texture upload.
- **GLDynamicTest**: renders a scene containing dynamic geometry.
- **GLComboTest**: renders a scene with various benchmark tests combined.

**Sequence Diagram**

This section details the primary sequence of messages and class communication within the application. The primary execution sequence of the application occurs during the running of a benchmark test.

**Run Test**

![Sequence Diagram for Running a Test](image)

Figure 18: Sequence Diagram for Running a Test
User Interface (UI) Design

A great deal of attention was paid to the application’s user interface in order to provide a user-friendly and intuitive experience. A good user interface is critical for all mobile applications. This is especially true for Android apps because of the diverse set of devices that an app must support. MOBILESpeed’s initial UI design was intended for larger, tablet sized devices (7 inches or greater) for best viewing, but later modified to support smaller cell phone sized devices as well. The key UI design elements that were taken into consideration included layout, navigation, theme, and compatibility.

A mockup of the complete application’s screen was created using an online diagramming resource (draw.io) as a starting point for the design. This mockup was used to brainstorm ideas for layouts, add desired app elements, and verify logical application navigation. The final design slightly deviated from the mockup for technical reasons.

Figure 19: User Interface Mockup

Android’s development site provides a very detailed UI design specification guide. For the design of MOBILESpeed, an effort was made to follow that guide at a high level.
Building the UI

To implement an application’s user interface, the Android framework allows developers to create layout structures, layout elements and widgets programmatically, or to declare them in XML. The preferred strategy is to utilize XML because it logically separates the UI and application code. This separation simplifies the process of modifying the look of an app without breaking the code. For MOBILESpeed, all user interfaces were constructed through XML and manipulated as needed programmatically.

The basic process for building a user interface in XML is as follows:

- Determine what UI elements (Buttons, text boxes, etc.) need to be on the screen.
- Decide on how the UI elements need to be organized in relation to one another.
- Choose a layout structure that will organize the UI elements as needed. The common layout structures include:
  - **Linear Layout**: organizes its child elements in a straight row. A horizontal or vertical orientation can be specified.
  - **Relative Layout**: organizes its child elements relative to one another. Each child element specifies its relative position.
  - **Table Layout**: organizes its child elements into rows and columns. The number of rows and columns is configurable.
  - **List View**: organizes its child elements in a vertical scrollable list. This is one of the most commonly used layout patterns in Android.
  - **GridView**: organizes its child elements in a 2D scrollable grid.
- Create the interface by building a hierarchy of UI elements.
MOBILESpeed uses the various layouts discussed above along with different widgets to create all the screens of the application that are presented in the section below. Before seeing the final UI, take a look at the following XML file which creates the “Home Screen” user interface:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1.5">
        <ImageView
            android:id="@+id/home_logo"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerInParent="true"
            android:contentDescription="@string/home_content_description"
            android:scaleType="centerInside"
            android:src="@drawable/home_logo"/>
        <TextView
            android:id="@+id/home_title"
            style="@style/floatingTextStyle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@id/home_logo"
            android:layout_centerInParent="true"
            android:gravity="center"
            android:paddingTop="@dimen/about_padding_topBottom"
            android:text="@string/app_name"
            android:textColor="@color/white"
            android:textSize="@dimen/about_app_name_textSize"/>
        <TextView
            android:id="@+id/about_subtitle"
            style="@style/floatingTextStyle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@id/home_logo"
            android:layout_centerInParent="true"
            android:gravity="center"
            android:paddingTop="@dimen/about_padding_topBottom"
            android:text="@string/about_subtitle"
            android:textColor="@color/white"
            android:textSize="@dimen/about_subtitle_textSize"/>
    </RelativeLayout>
</LinearLayout>
```
The root XML element of the Home screen is a vertical linear layout containing two nested layouts: a relative layout and horizontal linear layout. The root vertical linear layout is used to place the app logo and name above the two buttons. The nested relative
layout is used to position the app’s logo above the app’s name. The nested horizontal linear layout is used to position the two buttons next to one another. Once the position of all UI elements is set using the layouts, each element and layout has additional attributes that are defined within its XML tag.

Each screen presented in the next section is designed similarly to the Home screen, as a hierarchy of layouts and other UI elements.

Final Product Screenshots

The final user interface design is presented in this section along with an explanation of the capabilities and features of each screen.

Home Screen

The Home screen of the app is seen in the figure below and is presented to the user following the splash screen. From here, the user has the ability to navigate to any of the other screens via the navigation feature by either pressing the name of the tab on the navigation bar, or using the swipe gesture to slide the view over. The ability to run pre-defined tests or go to custom test creation is available from this screen via buttons.
The Tests screen of the app can be accessed in two ways: selecting the “TESTS” tab on the navigation bar, or by pressing the button titled “Create Custom Test” from the Home screen. A grid of available test types is displayed on this screen; the tests include: Points, Lines, Triangles, Texture, Lighting, Collision, Dynamic, Static, Bus Bandwidth.

Each cell in the grid represents a single test type. To create a custom benchmark test scene, select the cell(s) for the desired test type to activate it (indicated by a green border) and input the quantity of items to generate (Note: some tests do not support entering a quantity because it is not applicable for that test). Once the desired tests are selected, the user has two additional options to apply to the scene via check boxes:
- **Combine into single scene**: enabling this option takes each of the selected tests and renders it into one scene. Otherwise, each selected test is run in sequence rather than at once. It should be noted that the Static and Bus Bandwidth tests cannot be combined with other tests for technical reasons, and therefore will be ignored by this option.

- **Antialiasing (4x AA)**: enabling this option creates a rendering context with multisampling to improve visual quality at the cost of performance. If the device does not support multisampling then a standard rendering context is created.

By default, the single scene and antialiasing options are disabled. Once all selections have been made, press the “Run” button to start the benchmark.

![MOBILESpeed App Tests Screen](image)

**Figure 21: MOBILESpeed App Tests Screen**
Results Screen

The Results screen of the app can be accessed manually by selecting the “RESULTS” tab in the navigation bar. If the user views the Results screen prior to running a benchmark test to completion there will be no data to display (a message will indicate this to the user that a test needs to be ran). Upon running a benchmark to completion, the Results tab will automatically be displayed to the user.

This screen contains a scrollable list of test items. Each individual test that was part of the benchmark will be displayed as an item in its own row. A row of data is split into 2 columns: the first column is the type of test and the quantity of objects, and the second column is the frame rate metrics data. The results from the most recent test will remain on this screen until a new test is run in which case they will be overridden.
Device Info Screen

The Device Info screen of the app can be accessed by selecting the “DEVICE INFO” tab in the navigation bar. The purpose of this screen is to present information about the device that is relevant to graphics performance. The device hardware specs and software version information is obtained each time at app startup. The device information includes:

- **Model**: make and model number of the device.
- **OS Version**: operating system version.
- **CPU**: number of processing cores and the processor frequency (in GHz).
- **Memory**: amount of physical RAM (in GB).
- **Display Resolution**: screen resolution (WxH), refresh rate, screen pixel density.
- **GPU**: make and model number of the graphics processing unit.
- **3D Graphics**: OpenGL API version, texturing capability, supported extensions.

![Figure 23: MOBILESpeed App Device Info Screen](image-url)
About Screen

The About screen of the app is accessed by pressing the “ABOUT” button on the navigation bar. The purpose of this screen is to display app version, support contact, and copyright information as well as any other additional product information that may be needed in the future.

Figure 24: MOBILESpeed App About Screen
BENCHMARK SCENE DESIGN

While designing and developing benchmark tests, it is important to keep in mind graphics processing architecture in order to be able to adequately profile device performance. Creating a performance benchmark for a mobile GPU is different from creating one for a desktop computer GPU because of the tile-based rendering technique that most mobile GPUs utilize over immediate rendering, used by desktop GPUs. A recommended approach is to create a suite of microbenchmarks that isolate the elements of a scene. The elements of a scene include primitive geometry (points, lines, triangles) which form complex shapes, textures, blending, lighting and other effects, to name a few. Using this microbenchmark technique will allow the impact of adding each individual scene element to be measured. Once the cost of the scene elements is assessed individually, the allocation of things like polygon counts for various models can be accurately determined to balance aesthetics and performance.

MOBILESpeed provides a set of test scenes containing models constructed of various geometries, the ability to apply textures and lighting effects, render dynamic object models, test the effects of adding collision detection to a scene, as well as measure the memory bus bandwidth performance for uploading large texture data. MOBILESpeed also has the capability to combine all the scenes. The implementation detail of these scenes as well as the scene rendering architecture is discussed in the following sections.
Test Execution Model

![Test Execution Model Diagram](image)

A single instance of the scene renderer object is created to perform the entire OpenGL scene rendering. The renderer is also responsible for processing user inputs and constructing the desired test scenes. The renderer executes the following steps:

1. **Receive test parameters**
   - These are provided by the user via the test selection screen.

2. **Generate all the required scenes**
   - Depending on the number of tests selected and the quantity specified for the geometry, this step can take a noticeably long time.

3. **Run each test scene in sequence**
   - Every scene is executed for 20 seconds as metrics are gathered.

4. **Compute metrics**
   - After the completion of all the tests, frame rate metric data is computed and formatted for display.

To gain a better understanding of how the scene renderer works and the functions it performs, examine the code snippets below, some of which are the functions diagramed in the test execution model above.
The onSurfaceCreated method is inherited from the GLSurfaceView and is automatically called upon the creation of the rendering surface. It sets the OpenGL framebuffer clearing color here as it is not necessary to do this more than once. It then proceeds to initialize the benchmark test scenes.

```java
@Override
public void onSurfaceCreated(GL10 arg0, EGLConfig arg1) {

    // Set the Frame buffer clearing color
    GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    // Render the scenes and start the test
    buildScenes();
}
```

The onSurfaceChanged method is called once upon the creation of the rendering surface and each time the OS sends a configuration change message, which is usually when the orientation of the device changes. When this happens, the scene renderer must recreate the OpenGL projection matrix based on the new height and width of the screen.

```java
@Override
public void onSurfaceChanged(GL10 arg0, int width, int height) {

    GLES20.glViewport(0, 0, width, height);

    // Save values
    _screenWidth = width;
    _screenHeight = height;

    float ratio = (float) width / height;
    Matrix.frustumM(_projectionMatrix, 0, -ratio, ratio, -1.0f, 1.0f,
                    NEAR_PLANE, FAR_PLANE);
}
```

The onSurfaceChanged method is called once upon the creation of the rendering surface and each time the OS sends a configuration change message, which is usually when the orientation of the device changes. When this happens, the scene renderer must recreate the OpenGL projection matrix based on the new height and width of the screen.

```java
private void buildScenes() {

    // Create a list of tests to run and pass it
    // along to the renderer
}
```
_testList = new ArrayList<GLTest>();

_testList.add(new GLPointTest(10000, 21.0));
_testList.add(new GLPointTest(20000, 21.0));
_testList.add(new GLLineTest(10000, 20.0));
_testList.add(new GLLineTest(20000, 20.0));
_testList.add(new GLTriangleTest(10000, 20.0));
_testList.add(new GLTriangleTest(20000, 20.0));
_testList.add(new GLTextureTest(GLTextureTest.LO_RES, 20.0));
_testList.add(new GLTextureTest(GLTextureTest.HI_RES, 20.0));
_testList.add(new GLLightTest());
_testList.add(new GLCollisionTest(50));
_testList.add(new GLCollisionTest(100));
_testList.add(new GLDynamicTest(5000, 20.0));
_testList.add(new GLDynamicTest(10000, 20.0));
_testList.add(new GLStaticTest(_testParams.numStatic, 20.0));
_testList.add(new GLBusBandwidthTest());
_testList.add(new GLComboTest(_testParams));

// Start the clock and run the 1st test!
_stopwatch.start();
_running = true;
_currentTest = 0;
}

The buildScenes method is called during the scene render’s initialization sequence by the onSurfaceCreated method. Its role is to instantiate the classes that represent the benchmarks and add them to the list of tests to be executed.

private void drawScene(int index) {

    GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT |
                   GLES20.GL_DEPTH_BUFFER_BIT);

    // Set the camera position (View matrix)
    Matrix.setLookAtM(_viewMatrix, 0, 0.0f, 0.0f, 30.0f, // eye
                      0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f); // at
                      0.0f, 1.0f, 0.0f); // up

    // Combine the view and project matrix
    Matrix.multiplyMM(_MVPMatrix, 0, _projectionMatrix, 0, _viewMatrix, 0);

    // Create rotation transformation
float[] rotatedMtx = new float[16];
if(_angle > 359.9f) {
    _angle = 0.0f;
} else {
    _angle += ANGLE_INCREMENT;
}

Matrix.setRotateM(_rotationMatrix, 0, _angle, 0, 1, 0);
Matrix.multiplyMM(rotatedMtx, 0, _MVPMatrix, 0, _rotationMatrix, 0);

// Render the scene
_testList.get(index).draw(rotatedMtx, _rotationMatrix, _viewMatrix,
                        _projectionMatrix);

The drawScene method is part of the main drawing loop which is controlled by automatic
systems calls to an Android framework method called onDrawFrame. OnDrawFrame
iterates through all the tests created by buildScenes and passes each one as a parameter to
drawScene for execution. The drawScene method performs all necessary matrix
operations and calls the draw method of each test (the draw method structure will be
presented later on during the discussion of individual benchmark scenes).

Once test execution begins, the renderer can be interrupted; however partial
results will not be available. A complete test cycle is required for metrics to be gathered.
The renderer is also responsible for creating and maintaining the projection matrix and
updating the scene rotation matrix, both of which are provided to each scene during
rendering to create the spinning sphere effect. It should be noted that not all scenes use
the rotation matrix provided by the renderer since some object movements need to vary.

Now that the scene renderer’s architecture and implementation has been
presented, a look at the details of each benchmark test scene will follow.
Anatomy of a Scene

The design and implementation of every benchmark test scene contains common patterns used to provide a consistent look and feel to the code, enhancing the code’s modifiability, maintainability, and reusability. Each scene constructs all the geometry data one time during initialization (this is initiated by the renderer), and then repeatedly executes the draw calls on that data. During initialization, each test also creates its own shader program. For tests that do not require any special work done in the vertex or fragment shaders (simply a pass through shader), for example the Points, Lines and Triangles test, the same shader programs are compiled and linked for all. After the shaders are created, handles to the shader variables are also stored during initialization for later use during rendering.

Once all the setup and initialization is complete, a draw scene call is executed to render the scene. When a draw is executed, the standard sequence is as follows:

1. Make a timing measurement for the frame rate
2. Activate the shader program
3. Pass the color, vertex and transformation data to the vertex shader
4. Draw the object
5. Revert back any state changes made

The draw scene function is configured to be called continuously by the renderer through the use of Android’s “RENDERMODE_CONTINUOUSLY” state flag. This means that the scene will be re-rendered as fast as the system is capable of doing so. The alternative and much more resource friendly approach is to render the scene only when there is a
change in the drawing data. This rendering mode can’t be used for benchmarking purposes as it will invalidate the frame timing measurements.

To better understand the benchmark test scene code architecture, take a look at the following code snippets from the GLPointTest class:

```java
public GLPointTest(int count, double radius) {
    // Number of objects to render
    _numRequestedObj = count;
    _radius = radius;
    _vertexCount = 0;

    // Object rendering parameters
    int newCount = count / 3;
    double temp = Math.sqrt((double) newCount);

    // Generate 3 spheres
    _sphereVerticies = new ArrayList<GLVertex3D>();
    makeSphere(GLDrawUtils.makeCircularPoints((int)temp, radius, 0.0),
               radius, Math.PI / temp, GLColor.OUTER_SPHERE_COLOR);
    makeSphere(GLDrawUtils.makeCircularPoints((int)temp, radius - 3.0, 0.0),
               radius - 3.0, Math.PI / temp, GLColor.MID_SPHERE_COLOR);
    makeSphere(GLDrawUtils.makeCircularPoints((int)temp, radius - 6.0, 0.0),
               radius - 6.0, Math.PI / temp, GLColor.INNER_SPHERE_COLOR);

    // Make a primitive float array to be used as our FloatBuffer
    float floatArray[] = GLDrawUtils.toPrimitiveFloatArray(_sphereVerticies);

    // initialize vertex byte buffer for shape coordinates
    // (number of coordinate values * 4 bytes per float)
    ByteBuffer bb = ByteBuffer.allocateDirect(floatArray.length * Consts.BYTES_PER_FLOAT);
    bb.order(ByteOrder.nativeOrder());

    // use the device hardware's native byte order
    _vertexBuffer = bb.asFloatBuffer();

    // add the coordinates to the FloatBuffer
    _vertexBuffer.put(floatArray);
}
```
// Create the Shader program
GLSLShader program = new GLSLShader();
int vsId = program.loadShader(GLSLShader.VERTEX, "shader/BaseVertexShader.glsl");
int fsId = program.loadShader(GLSLShader.FRAGMENT, "shader/BaseFragmentShader.glsl");
_shaderProgram = program.createProgram(vsId, fsId);
}

The constructor of the GLPointTest class receives the quantity of point objects to render and uses the makeSphere method to produce the concentric sphere drawing effect. Then an array of floating point values is created to contain the vertices representing a location in 3D space where the point objects are to be drawn. Lastly, the vertex and fragment shader source code is read in from a file, then compiled and linked to create the shader program for this test. The source for the makeSphere method as well as the shaders is available below.

private void makeSphere(ArrayList<PointF> location, double radius, double interval, float color[]) {
    for(double i = -Consts.HALF_PI; i < Consts.HALF_PI; i += interval) {
        double y = radius * Math.sin(i);
        for(int itr = 0; itr < location.size(); itr++) {

            double x = location.get(itr).x * Math.cos(i);
            double z = location.get(itr).y * Math.cos(i);

            GLVertex3D currVertex = new GLVertex3D(
                new GLPosition3D((float)x, (float)y, (float)z),
                new GLColor(color[0], color[1], color[2], color[3]));

            _sphereVerticies.add(currVertex);
            _vertexCount++;
        }
    }
}
The vertex shader receives the Model-View-Projection matrix along with position and color data from the main Java program, transforms the input position using the MVP, scales the point size to cover 2 pixels and outputs the results of those operations to be used in the subsequent stages of processing pipeline. The vertex color data is also passed through this stage to the fragment shader. The fragment shader receives the color data and applies the final color to the object being rendered. Once initialization of the GLPointTest is complete, the draw method is called to render the scene.

```java
@Override
public void draw(float[] mvpMatrix, float[] modelMatrix, float[] viewMatrix, float[] projMatrix) {
    if (!_stopwatch.isRunning()) { _stopwatch.start(); }

    // Store draw time of the last frame
    _currTime  = _stopwatch.elapsedTime();
    _deltaTime = _currTime - _lastTime;
    _drawTimes.add(new Float(_deltaTime));
    _lastTime = _currTime;

    // Add program to OpenGL ES environment
    GLES20.glUseProgram(_shaderProgram);

    // Prepare position data
    _PositionHandle = GLES20.glGetAttribLocation(_shaderProgram, "a_Position");
```
GLES20.glEnableVertexAttribArray(_PositionHandle);
GLES20.glVertexAttribPointer(_PositionHandle,
    GLPosition3D.ELEMENTS_PER_VERTEX,
    GLES20.GL_FLOAT,
    false,
    GLVertex3D.VERTEX_STRIDE,
    _vertexBuffer.position(0));

// Prepare color data
_ColorHandle = GLES20.glGetAttribLocation(_shaderProgram, "a_Color");
GLES20.glEnableVertexAttribArray(_ColorHandle);
GLES20.glVertexAttribPointer(_ColorHandle,
    GLColor.ELEMENTS_PER_VERTEX,
    GLES20.GL_FLOAT,
    false,
    GLVertex3D.VERTEX_STRIDE,
    _vertexBuffer.position(3));

// Get handle to shape's transformation matrix
_MVPMatrixHandle = GLES20.glGetUniformLocation(_shaderProgram, "uMVPMatrix");

// Pass the projection and view transformation to the shader
GLES20.glUniformMatrix4fv(_MVPMatrixHandle, 1, false, mvpMatrix, 0);

// Draw the object
GLES20.glDrawArrays(GLES20.GL_POINTS, 0, _vertexCount);

// Disable vertex array
GLES20.glDisableVertexAttribArray(_PositionHandle);
GLES20.glDisableVertexAttribArray(_ColorHandle);
}

The draw method is called by the drawScene method in the SceneRender class, which was reviewed earlier in this section. Each time draw is called, it must activate the shader program in which it wants to use (multiple shader programs can be created but only one active at a time). Then handles to the variables that reside in the vertex and fragment shaders are obtained in order to transfer data between the main and shader program.
Finally, the `glDrawArrays` API call is made to render the data. The draw method exists in all classes that implement the GLTest interface.

All OpenGL test classes follow a similar pattern to the one presented above. The remainder of this section will provide technical details regarding every benchmark test scene that is available in MOBILESpeed.

Scene 1 - Points

![Figure 26: Rendering Scene of 20,000 Points](image)

This scene renders 3 concentric spheres of varying radii utilizing the `GL_POINTS` primitive. The quantity of points rendered is configurable by the user.

Scene implementation details:

- Objective: measure drawing performance of the `GL_POINTS` primitive.
● Geometry: static (vertices initialized once)

● Shaders:
  ○ Vertex shader scales point size to be 2 pixels instead of the default 1 pixel.
  ○ Fragment shader applies specified color to each point/vertex.

Scene 2 - Lines

Figure 27: Rendering Scene of 20,000 Lines

This scene renders 3 concentric spheres of varying radii utilizing the GL_LINES primitive. The quantity of lines rendered is configurable by the user.

Scene implementation details:

● Objective: measure drawing performance of the GL_LINES primitive.

● Geometry: static (vertices initialized once)
● Shaders:
  ○ Vertex shader performs vertex transformation using the MVP matrix.
  ○ Fragment shader applies specified color to each point/vertex.

Scene 3 - Triangles

![Diagram of 3 concentric spheres rendered with GL_TRIANGLES primitive.](image)

Figure 28: Rendering Scene of 10,000 Triangles

This scene renders 3 concentric spheres of varying radii utilizing the GL_TRIANGLES primitive. The quantity of triangles rendered is configurable by the user.

Scene implementation details:

● Objective: measure drawing performance of the GL_TRIANGLES primitive.
● Geometry: static (vertices initialized once)
• Shaders:
  ○ Vertex shader performs vertex transformation using the MVP matrix.
  ○ Fragment shader applies specified color to each point/vertex.

Scene 4 - Texture

Figure 29: Rendering Scene of 512x512 Textures

This scene renders three textured cubes of various sizes. The quantity of textured cubes rendered is not configurable, however the resolution of the images used for the texture is configurable by the user. The available texture resolutions for this test are 128x128 pixels, 512x512 pixels, and 2048x2048 pixels. Keep in mind that the maximum supported texture size is hardware dependent but on most devices the max size is typically 4096x4096 pixels. Android texture usage best practices discourages using non-
compressed textures at higher resolutions than the screen space it is being displayed on, however for benchmarking purposes compression will be ignored since the purpose is to simulate a stressful processing scenario and exercise the memory bandwidth.

Scene implementation details:

- Objective: measure drawing performance of textures of various resolutions.
- Geometry: static (vertices initialized once, texture uploaded once)
- Shaders:
  - Vertex shader performs vertex transformation using the MVP matrix.
  - Fragment shader performs the texture processing. It receives the texture coordinates and produces a texel (a single pixel in the texture image) value that is applied to the final fragment result.
This scene renders three textured cubes of various sizes with the addition of a revolving light source to light the cube’s surfaces. The quantity of textured cubes and light sources is not configurable by the user. The elements of this scene are similar to the Texture benchmark test scene, with exception of the light source which is intended to add a more complex shader in order to produce the lighting effects.

The scene implements a combination of ambient and diffuse lighting effects with the origin of the light source being a point source. The algorithm used in the fragment shader to produce the lighting effect is as follows:
1. Compute the angle between each surface of the cube and the light source as well as the distance of the surface and the light.

2. Compute the attenuation factor of the light for determining the luminosity at each point on the cube’s surface.

3. Using the texture color and illumination computed in the previous step, compute the final fragment color.

Scene implementation details:

- Objective: measure drawing performance of textured cubes with lighting effects.
- Geometry: static (vertices initialized once, texture uploaded once)
- Shaders:
  - Vertex shader performs vertex transformation using the MVP matrix. It also converts the surface normal vectors from model to eye/camera space.
  - Fragment shader performs the texture processing. It receives the texture coordinates and produces a texel (a single pixel in the texture image) value that is applied to the final fragment result. Once the texture is processed, the fragment shader computes the lighting direction vector to determine the illumination intensity which is factored into the final fragment’s color.
Scene 6 - Collision

Figure 31: Rendering Scene of Colliding Spheres

This scene renders a user defined quantity of spheres moving through space from different starting points at various velocities. The purpose of this test is to benchmark the additional overhead of computing object collisions every frame. Each time a sphere collides with another, the color of the intersecting spheres is indicated visually with the color red, while all non-colliding spheres remain the default blue. For this test, the colliding bodies are in 2-dimensional space and the collision is detected by checking if the distance between the center points of each sphere is less than twice the radius. The algorithm implemented runs in $n^2$ time.

Scene implementation details:
- Objective: measure drawing performance with the cost of performing collision
detection on objects.
- Geometry: static (vertices defining each sphere are initialized once)
- Shaders:
  - Vertex shader performs vertex transformation using the MVP matrix.
  - Fragment shader applies specified color to each point/vertex.

Scene 7 - Dynamic

![Figure 32: Rendering Scene of Dynamic Spheres](image)

This scene renders 3 concentric spheres of varying radii utilizing the GL_POINTS
primitive in a dynamic fashion, meaning the geometry is periodically rebuilt. Every two
seconds the vertices and colors that make up the spheres are created again, unlike the
non-dynamic version of this test where all the geometry and color information is initialized one time into a vertex buffer. Doing this simulates a dynamic scene where the environment is continuously changing based on object interaction. If the device being benchmarked is fast enough then the transition between the changing states of the spheres will be more fluid, otherwise a slight pause will be noticed as the geometry and color information is rebuilt. The change in the objects rendered is indicated visually through varying sizes and colors, and the quantity rendered is configurable by the user.

Scene implementation details:

- Objective: measure drawing performance when using dynamic geometry.
- Geometry: dynamic (new vertices created periodically)
- Shaders:
  - Vertex shader scales point size to be 2 pixels instead of the default 1 pixel.
  - Fragment shader applies specified color to each point/vertex.
Scene 8 - Static

Figure 33: Rendering Scene of Static Frames

This scene renders the same exact frame 100 times consecutively and is based off the Triangles test, which renders 3 concentric spheres of varying radii utilizing the GL_TRIANGLES primitive to compose a single frame. The quantity of triangles rendered is configurable by the user but the consecutive frames rendered count is not.

This test was specifically designed to test PowerVR GPUs although conceptually it should apply to any GPU that utilizes a similar architecture. It was implemented according to a benchmarking guide provided by Imagination Technologies, the makers of PowerVR graphics processors. The PowerVR guide describes several techniques to use in order to accurately isolate GPU processing and measure performance. The first step is to
ensure that V-Sync is disabled for the device under test as this will force GPU and CPU synchronization to occur and reduce the number of frame processed. This option is not commonly available on devices and most guides suggest that doing this is difficult or impossible. As a workaround to this limitation, the Static test implements a loop in its main drawing routine to submit the same frame repeatedly (100 times) for rendering to the GPU rather than once per draw call by the system. The second step is to verify that no other processes are concurrently using GPU resources. Since the test runs in full screen mode and all other background applications are terminated using the Android’s process manager, there should be no GPU resource contention. The third step is to warm-up, or initialize the assets to be used, which means forcing the graphics driver to perform operations like texture or geometry uploading instead of deferring the process to a later time when the resources will actually be used. The Static test accomplishes this by performing multiple passes through the draw sequence when collecting timing measurements. The fourth step is the actual test execution which is done by first making a time measurement for the start time, rendering the same frame multiple times by calling the glDrawArrays function, then calling the glReadPixels function to acquire the framebuffer contents which flushes the command buffer and forces all submitted rendering commands to process to completion. Finally, make another time measurement after glReadPixels completes and compute the average time taken to render a frame by dividing the elapsed time by the number of frames that were rendered.
The Static test implementation described above deviates slightly from the steps outlined in the PowerVR guide for technical reasons as the guide states to manually force a framebuffer swap after `glReadPixels` completes execution, however the call to perform the buffer swap is done automatically by the system since the application uses a `GLSurfaceView` object to establish the rendering context.

The Static test is the only benchmark created that strictly focuses on measuring GPU performance. Utilizing this benchmarking method appears to return consistent and reliable results on all devices that were tested, even on non-PowerVR architectures.

Scene implementation details:

- **Objective**: measure the average cost of rendering a single, identical frame in a scene containing no dynamic parts and consistent API calls.
- **Geometry**: static (vertices initialized once). Same frame rendered repeatedly.
- **Shaders**:
  - Vertex shader performs vertex transformation using the MVP matrix.
  - Fragment shader applies specified color to each point/vertex.
Scene 9 - Bus Bandwidth

Figure 35: Rendering Blank Scene to Test Bus Bandwidth

This scene does no rendering of graphical content. The purpose of this test is to benchmark the time required to upload texture data to video memory. Bus performance is a fairly common bottleneck of graphics system because there is a finite data transfer rate between components. The GPUs utilization is highly dependent on the rate in which it receives data.

Scene implementation details:

- Objective: measure bus bandwidth performance for large data transfer.
- Geometry: none (only a simple texture containing the test name is rendered)
- Shaders:
- Vertex shader does no special processing.
- Fragment shader processes the texture that contains the “Bus bandwidth test running…” text display on the screen.

**Scene 10 - Combined**

Figure 36: Rendering Scene of Combined Elements

This scene combines and renders the various benchmark test scenes available in the application. The user has the ability to select any combination of individual tests to be combined into this single scene. The only tests that are an exception and cannot be combined with others are the Static and Bus Bandwidth tests due to certain technical differences in the implementation of those scenes compared to all others. The purpose of
this test is to allow all the elements of a typical scene (polygons, lighting, textures, collision detection, etc.) to come together and measure performance as a whole.

Scene implementation details:

- Objective: measure drawing performance of a scene containing various graphical elements all at once.
- Geometry: static and dynamic (some vertices initialized once, the vertices for the dynamic objects are re-created repeatedly).
- Shaders:
  - Vertex shaders vary based on which test is combined. See the individual test’s shader section for more information.
  - Fragment shaders vary based on which test is combined. See the individual test’s shader section for more information.

Design Challenges

The most notable challenge when designing the benchmark scenes was implementing them under Android’s heap memory limitations. Since the quantity of objects rendered for most tests scenes is configurable, the initial implementation did not place an upper limit on the user input, however in most cases input values above a few hundred thousand caused the application to either crash or take an unreasonably long time to generate the scene. The application crashing issue became more apparent when the scenes were set to be combined. After some analysis, two solutions were decided upon: an appropriate quantity limit was placed on each available test for application robustness, and Android’s large heap option was enabled.
PROJECT ENVIRONMENT

The Eclipse Integrated Development Environment (IDE) along with the Android Development Tools (ADT) plugin is the standard development environment for building Android applications. More recently, Google has formally transitioned to promoting and fully supporting use of its custom built IDE, Android Studio (which is based on the popular IntelliJ IDE). This section will present the details for creating the MOBILESpeed Android project using the Eclipse/ADT development environment.

Required Software

At a minimum, the following items need to be installed on the development system in order to create a working project. For the MOBILESpeed application, these were installed on a Windows 7 system, however all are also available for other operating systems.

- Java Runtime Environment
- Java JDK 6
- Eclipse IDE for Android Developers (Android SDK)
- OpenGL ES 2.0 Library

Installation

The installation process is straightforward and similar regardless of the operating system being used. The first step is to download the software listed above. Once the required software is obtained, the installation procedure can begin.

The Java Runtime Environment (JRE) is commonly installed on all operating systems. To verify it’s installed or if its installation is required, go to Oracle’s website
and follow the instructions on the page. The link below will either confirm that an up-to-
date version of JRE exists, or present an option to install/update the JRE version.

https://www.java.com/en/download/installed.jsp

Ensure that the JRE is installed and up-to-date before proceeding with the installation.

The Java JDK software can be obtained from Oracle’s official website. Keep in
mind that Java may already be installed on your system so verify this beforehand to avoid
reinstallation. To check for an installed JDK version, open a Command Prompt on
Windows and input the following command:

C:> for %i in (javac.exe) do @echo. %~$PATH:i

If the resulting output from the command is a directory similar to “C:\Program
Files\Java\jdk1.6.0\bin\javac.exe” then there is no need for JDK installation.

MOBILESpeed was developed using JDK 6, however the latest JDK version is usually
recommended to be used and can be found here:

www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads

Once the Java JRE and JDK have been installed, proceed to the Eclipse ADT installation.

Android’s website provides a single installer that contains the appropriate version
of Eclipse and the Android ADT that can be used for simple installation on Windows:


The main SDK page contains the download all option as well as additional installation
resources in case installation of only specific components is desired. For MOBILESpeed,
the download all option is sufficient.

The OpenGL ES library is provided automatically by the Android framework
given that a specific minimum API is targeted by the application. MOBILESpeed
requires a minimum API level of 14, with a targeted version of API level 19. For OpenGL ES 2.0 functionality, API level 8 or higher is sufficient.

**Project Creation**

Once Eclipse is installed, launch the application and perform the steps provided in this section to create and configure the MOBILESpeed project environment.

1. Create new Android application project.

![Creating a New Android Application Project](image-url)

Figure 37: Creating a New Android Application Project
2. Set the application name and target API level

![New Android Application Project Details](image)

**Figure 38: New Android Application Project Details**

In this prompt, the app name is selected and the minimum and target API levels that the application will support are determined. This is an important step during the project creation because it will prevent or allow the app from using certain features.

MOBILESpeed requires a minimum API level of 14, with a target of 19. Fill in the required information and proceed to the next step.

3. Set additional application properties
On this screen, deselect the “Create custom launcher icon” option as this does not need to be done now. Leave all other options as their default and proceed to the next step.

4. Create the main Activity
Select “Blank Activity” and proceed to the next step.

5. Name the main Activity and its layout XML file.

![Android Application Configuration - 3](image)

It is recommended to leave the main Activity class named “MainActivity” but this isn’t required. For MOBILESpeed, the standard naming convention of using “MainActivity” for the class and “activity_main” for the XML layout file were kept. Select finish to create the project.
The Eclipse workspace now displays the newly created MOBILESpeed project (see figure above). The project is ready for development once the AndroidManifest.xml file is updated to include a few MOBILESpeed specific features. Open the manifest XML file located in the project tree (left-hand side of figure above) and add the following lines:

```xml
<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true" />
```

The manifest declaration is required to ensure the app targets OpenGL ES API level 2.0.

**Emulator/Hardware Test Configuration**

As the project develops, testing on an Android device is required. There are two ways the application can be tested: using a virtual device emulator or using a physical Android device. The latter method is preferred because on most systems the emulator exhibits noticeable performance degradation, making it difficult to effectively perform
testing. The ideal testbed is using multiple physical devices, each running different API versions, having different screen sizes and screen pixel densities.

To configure a physical device for testing, launch Eclipse and open the project. Once the project is open, plug in the device via the USB port. The device should be automatically detected at this point. Open any java class file within the project and on the top toolbar in Eclipse, press the “Run as” button (shown in screenshot below).

![Figure 43: How to Run an Application in Eclipse](image)

After pressing the run button, the bottom status bar and the LogCat tab in Eclipse should indicate that the project is installing and launching. The application should now be displayed on the connected physical device and ready for testing.

To configure the emulator for testing, select the “Run as” button similar to running on a physical device. Since no physical device is connected, a message prompting to add a new “Android Virtual Device” will be displayed. Press “Ok” to display the virtual device configuration screen (seen in the screenshot below).
Select the desired virtual device displayed in the bottom pane and press “Start..” to run the device emulator. Once the emulator is running, the app will launch.

A notable drawback of the emulator is the unavailability of certain system calls. MOBILESpeed’s Device Info screen is untestable on the emulator because of the API and hardware specific calls it makes to retrieve information about the device. This reason and the noticeable sluggishness introduced by the emulation process is why a physical device is preferred.

The benefit of the emulator is the ability it provides to view the app’s layout design on many devices to ensure a uniform look and feel on all devices. An important component of a good mobile app is how it supports various devices.
Version Control

The project content for MOBILESpeed is stored online in a GitHub repository. Once a GitHub repository is created, Eclipse offers a plugin to connect to the repository and also integrate commonly used features of Git directly into the Eclipse IDE. To download and install the Git plugin, perform the following steps:

1. In Eclipse, go to Help > Eclipse Marketplace
2. In the search bar, type in “Git” and press the Go button.
3. In the search results, look for “EGit - Eclipse Git Team Provider”

![EGit - Git Team Provider](image)

Select the “Install” option (“Update” is seen here since EGit is already installed).

4. After installation, the Git toolbar should be available in Eclipse. It features various options to sync the project files with the GitHub repository.

![Integrated Git Toolbar in Eclipse](image)

This concludes the project environment configuration tutorial. The Eclipse project is now setup and ready for app development.
DEVICE PERFORMANCE DATA

A total of six Android devices of various classes were tested using MOBILESpeed. These devices ranged from low to high-end products. This section details the hardware specifications and performance results for all devices tested.

Devices Under Test

The following mobile Android devices were tested using MOBILESpeed: Acer Iconia A1, Samsung Galaxy Tab 2, Samsung Galaxy Note 3, HTC One, HTC One V, and LG G3.

Hardware Specifications

Each column in the hardware specs table contains the following information:

- **Graphics**: graphics processing unit, supported OpenGL version, number of texture mapping units (TMU), maximum size for a texture image.
- **Display**: physical screen size, resolution in pixels (WxH), screen refresh rate, screen pixel density.
- **CPU**: clock frequency, number of physical processing cores
- **Memory**: amount of physical RAM installed.

<table>
<thead>
<tr>
<th>Device</th>
<th>Graphics</th>
<th>Display</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acer Iconia A1 (A1-810)</td>
<td>PowerVR SGX 544MP</td>
<td>7.9 inches</td>
<td>1.2 GHz</td>
<td>1.0 GB</td>
</tr>
<tr>
<td></td>
<td>OpenGL ES 2.0</td>
<td>1024x720</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 Texture units</td>
<td>58.3Hz refresh</td>
<td>4 cores</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4096x4096 Texture size</td>
<td>160 dpi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Type</td>
<td>Model</td>
<td>GPU</td>
<td>Display Size</td>
<td>CPU Frequency</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------------</td>
<td>----------------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Samsung Galaxy Tab 2</td>
<td>GT-P5113</td>
<td>PowerVR SGX 540</td>
<td>10.1 inches</td>
<td>1.0 GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpenGL ES 2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 Texture units</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2048x2048 Texture size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTC One V</td>
<td></td>
<td>Adreno 205</td>
<td>3.7 inches</td>
<td>1.0 GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpenGL ES 2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 Texture units</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4096x4096 Texture size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Samsung Galaxy Note 3</td>
<td>SM-N900A</td>
<td>Adreno 330</td>
<td>5.7 inches</td>
<td>2.3 GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpenGL ES 3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 Texture units</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4096x4096 Texture size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTC One</td>
<td>HTC6500LVW</td>
<td>Adreno 320</td>
<td>4.7 inches</td>
<td>1.7 GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpenGL ES 3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 Texture units</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4096x4096 Texture size</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 4: Hardware Specs for Devices Under Test

Results

This section contains frame rate performance data for each device. Each available benchmark test was run at a normal and high level of intensity.

Acer Iconia A1

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Object Count</th>
<th>Average FPS</th>
<th>Average Frame Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>20,000</td>
<td>58.46</td>
<td>17.11 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>43.20</td>
<td>23.15 ms</td>
</tr>
<tr>
<td>Lines</td>
<td>20,000</td>
<td>24.34</td>
<td>41.08 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>5.54</td>
<td>180.45 ms</td>
</tr>
<tr>
<td>Triangles</td>
<td>20,000</td>
<td>58.17</td>
<td>17.19 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>19.19</td>
<td>52.11 ms</td>
</tr>
<tr>
<td>Texture</td>
<td>512</td>
<td>58.33</td>
<td>17.14 ms</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>49.29</td>
<td>20.29 ms</td>
</tr>
<tr>
<td>Lighting</td>
<td>N/A</td>
<td>58.14</td>
<td>17.20 ms</td>
</tr>
<tr>
<td>Collision</td>
<td>100</td>
<td>57.54</td>
<td>17.38 ms</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>4.48</td>
<td>223.25 ms</td>
</tr>
<tr>
<td>Dynamic</td>
<td>8,000</td>
<td>52.63</td>
<td>19.00 ms</td>
</tr>
</tbody>
</table>
## Table 5: Frame Rate Results for Acer Iconia A1

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Object Count</th>
<th>Average FPS</th>
<th>Average Frame Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>50,000</td>
<td>30.16</td>
<td>33.16 ms</td>
</tr>
<tr>
<td></td>
<td>125,000</td>
<td>11.93</td>
<td>83.83 ms</td>
</tr>
<tr>
<td>Bus Bandwidth</td>
<td>160 MB</td>
<td>0.71</td>
<td>1411.00 ms</td>
</tr>
<tr>
<td>Combined</td>
<td>(Normal)</td>
<td>14.03</td>
<td>71.26 ms</td>
</tr>
<tr>
<td></td>
<td>(High)</td>
<td>0.86</td>
<td>1162.82 ms</td>
</tr>
</tbody>
</table>

## Samsung Galaxy Tab 2

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Object Count</th>
<th>Average FPS</th>
<th>Average Frame Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>20,000</td>
<td>59.60</td>
<td>16.78 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>19.52</td>
<td>51.22 ms</td>
</tr>
<tr>
<td>Lines</td>
<td>20,000</td>
<td>14.63</td>
<td>68.34 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>2.02</td>
<td>495.90 ms</td>
</tr>
<tr>
<td>Triangles</td>
<td>20,000</td>
<td>58.96</td>
<td>16.96 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>6.01</td>
<td>166.32 ms</td>
</tr>
<tr>
<td>Texture</td>
<td>512</td>
<td>59.74</td>
<td>16.74 ms</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>47.57</td>
<td>21.02 ms</td>
</tr>
<tr>
<td>Lighting</td>
<td>N/A</td>
<td>59.44</td>
<td>16.82 ms</td>
</tr>
<tr>
<td>Collision</td>
<td>100</td>
<td>58.99</td>
<td>16.95 ms</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>3.64</td>
<td>275.08 ms</td>
</tr>
<tr>
<td>Dynamic</td>
<td>8,000</td>
<td>47.73</td>
<td>20.95 ms</td>
</tr>
<tr>
<td></td>
<td>50,000</td>
<td>8.00</td>
<td>124.93 ms</td>
</tr>
<tr>
<td>Static</td>
<td>50,000</td>
<td>16.04</td>
<td>62.34 ms</td>
</tr>
<tr>
<td></td>
<td>125,000</td>
<td>4.65</td>
<td>215.11 ms</td>
</tr>
<tr>
<td>Bus Bandwidth</td>
<td>160 MB</td>
<td>0.44</td>
<td>2260.88 ms</td>
</tr>
</tbody>
</table>
Table 6: Frame Rate Results for Samsung Galaxy Tab 2

**HTC One V**

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Object Count</th>
<th>Average FPS</th>
<th>Average Frame Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>20,000</td>
<td>60.21</td>
<td>16.61 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>19.65</td>
<td>50.88 ms</td>
</tr>
<tr>
<td>Lines</td>
<td>20,000</td>
<td>29.27</td>
<td>34.17 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>6.47</td>
<td>154.53 ms</td>
</tr>
<tr>
<td>Triangles</td>
<td>20,000</td>
<td>46.18</td>
<td>21.66 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>12.71</td>
<td>78.67 ms</td>
</tr>
<tr>
<td>Texture</td>
<td>512</td>
<td>60.21</td>
<td>16.61 ms</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>59.38</td>
<td>16.84 ms</td>
</tr>
<tr>
<td>Lighting</td>
<td>N/A</td>
<td>58.03</td>
<td>17.23 ms</td>
</tr>
<tr>
<td>Collision</td>
<td>100</td>
<td>35.36</td>
<td>28.28 ms</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>1.35</td>
<td>743.38 ms</td>
</tr>
<tr>
<td>Dynamic</td>
<td>8,000</td>
<td>54.46</td>
<td>18.36 ms</td>
</tr>
<tr>
<td></td>
<td>50,000</td>
<td>14.44</td>
<td>69.25 ms</td>
</tr>
<tr>
<td>Static</td>
<td>10,000</td>
<td>Would not run</td>
<td>Would not run</td>
</tr>
<tr>
<td></td>
<td>125,000</td>
<td>Would not run</td>
<td>Would not run</td>
</tr>
<tr>
<td>Bus Bandwidth</td>
<td>160 MB</td>
<td>0.41</td>
<td>2412.50 ms</td>
</tr>
<tr>
<td>Combined</td>
<td>(Normal)</td>
<td>9.69</td>
<td>103.23 ms</td>
</tr>
<tr>
<td></td>
<td>(High)</td>
<td>0.46</td>
<td>2181.75 ms</td>
</tr>
</tbody>
</table>

Table 7: Frame Rate Results for HTC One V
### Samsung Galaxy Note 3

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Object Count</th>
<th>Average FPS</th>
<th>Average Frame Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>20,000</td>
<td>59.51</td>
<td>16.80 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>59.64</td>
<td>16.77 ms</td>
</tr>
<tr>
<td>Lines</td>
<td>20,000</td>
<td>59.39</td>
<td>16.84 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>18.02</td>
<td>55.50 ms</td>
</tr>
<tr>
<td>Triangles</td>
<td>20,000</td>
<td>59.91</td>
<td>16.69 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>29.40</td>
<td>34.02 ms</td>
</tr>
<tr>
<td>Texture</td>
<td>512</td>
<td>59.14</td>
<td>16.91 ms</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>59.18</td>
<td>16.90 ms</td>
</tr>
<tr>
<td>Lighting</td>
<td>N/A</td>
<td>59.63</td>
<td>16.77 ms</td>
</tr>
<tr>
<td>Collision</td>
<td>100</td>
<td>59.36</td>
<td>16.85 ms</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>7.76</td>
<td>128.82 ms</td>
</tr>
<tr>
<td>Dynamic</td>
<td>8,000</td>
<td>54.14</td>
<td>18.47 ms</td>
</tr>
<tr>
<td></td>
<td>50,000</td>
<td>37.45</td>
<td>26.70 ms</td>
</tr>
<tr>
<td>Static</td>
<td>50,000</td>
<td>57.27</td>
<td>17.46 ms</td>
</tr>
<tr>
<td></td>
<td>125,000</td>
<td>17.79</td>
<td>56.23 ms</td>
</tr>
<tr>
<td>Bus Bandwidth</td>
<td>160 MB</td>
<td>2.57</td>
<td>389.59 ms</td>
</tr>
<tr>
<td>Combined</td>
<td>(Normal)</td>
<td>36.65</td>
<td>27.28 ms</td>
</tr>
<tr>
<td></td>
<td>(High)</td>
<td>3.73</td>
<td>268.35 ms</td>
</tr>
</tbody>
</table>

Table 8: Frame Rate Results for Samsung Galaxy Note 3

### HTC One

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Object Count</th>
<th>Average FPS</th>
<th>Average Frame Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>20,000</td>
<td>58.61</td>
<td>17.06 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>59.09</td>
<td>16.92 ms</td>
</tr>
<tr>
<td>Test Name</td>
<td>Object Count</td>
<td>Average FPS</td>
<td>Average Frame Time</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Lines</td>
<td>20,000</td>
<td>58.39</td>
<td>17.13 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>12.74</td>
<td>78.50 ms</td>
</tr>
<tr>
<td>Triangles</td>
<td>20,000</td>
<td>58.74</td>
<td>17.02 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>20.17</td>
<td>49.59 ms</td>
</tr>
<tr>
<td>Texture</td>
<td>512</td>
<td>58.46</td>
<td>17.10 ms</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>43.81</td>
<td>22.83 ms</td>
</tr>
<tr>
<td>Lighting</td>
<td>N/A</td>
<td>43.89</td>
<td>22.78 ms</td>
</tr>
<tr>
<td>Collision</td>
<td>100</td>
<td>58.99</td>
<td>16.95 ms</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>6.38</td>
<td>156.73 ms</td>
</tr>
<tr>
<td>Dynamic</td>
<td>8,000</td>
<td>53.57</td>
<td>18.67 ms</td>
</tr>
<tr>
<td></td>
<td>50,000</td>
<td>32.37</td>
<td>30.89 ms</td>
</tr>
<tr>
<td>Static</td>
<td>50,000</td>
<td>50.30</td>
<td>19.88 ms</td>
</tr>
<tr>
<td></td>
<td>125,000</td>
<td>50.92</td>
<td>19.64 ms</td>
</tr>
<tr>
<td>Bus Bandwidth</td>
<td>160 MB</td>
<td>1.41</td>
<td>710.25 ms</td>
</tr>
<tr>
<td>Combined</td>
<td>(Normal)</td>
<td>24.17</td>
<td>41.37 ms</td>
</tr>
<tr>
<td></td>
<td>(High)</td>
<td>2.79</td>
<td>358.04 ms</td>
</tr>
</tbody>
</table>

Table 9: Frame Rate Results for HTC One

**LG G3**

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Object Count</th>
<th>Average FPS</th>
<th>Average Frame Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>20,000</td>
<td>49.99</td>
<td>20.00 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>49.73</td>
<td>20.11 ms</td>
</tr>
<tr>
<td>Lines</td>
<td>20,000</td>
<td>49.43</td>
<td>20.23 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>12.60</td>
<td>79.39 ms</td>
</tr>
<tr>
<td>Triangles</td>
<td>20,000</td>
<td>50.02</td>
<td>19.99 ms</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>30.53</td>
<td>32.76 ms</td>
</tr>
<tr>
<td></td>
<td>Val1</td>
<td>Val2</td>
<td>Val3</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>Texture</td>
<td>512</td>
<td>50.00</td>
<td>20.00 ms</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>49.99</td>
<td>20.00 ms</td>
</tr>
<tr>
<td>Lighting</td>
<td>N/A</td>
<td>49.99</td>
<td>20.00 ms</td>
</tr>
<tr>
<td>Collision</td>
<td>100</td>
<td>50.01</td>
<td>20.00 ms</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>10.21</td>
<td>97.98 ms</td>
</tr>
<tr>
<td>Dynamic</td>
<td>8,000</td>
<td>46.74</td>
<td>21.39 ms</td>
</tr>
<tr>
<td></td>
<td>50,000</td>
<td>34.39</td>
<td>29.08 ms</td>
</tr>
<tr>
<td>Static</td>
<td>50,000</td>
<td>66.63</td>
<td>15.01 ms</td>
</tr>
<tr>
<td></td>
<td>125,000</td>
<td>Would not run</td>
<td>Would not run</td>
</tr>
<tr>
<td>Bus Bandwidth</td>
<td>160 MB</td>
<td>1.85</td>
<td>542.00 ms</td>
</tr>
<tr>
<td>Combined</td>
<td>(Normal)</td>
<td>32.22</td>
<td>31.04 ms</td>
</tr>
<tr>
<td></td>
<td>(High)</td>
<td>4.06</td>
<td>246.04 ms</td>
</tr>
</tbody>
</table>

Table 10: Frame Rate Results for LG G3

Analysis

After running the test scenarios on each device and gathering the frame rate metrics in terms of frame time, some general observations can be made regarding the performance results of each device:

- Devices with superior hardware specifications perform better in most test cases.
- Degradation in performance is not proportional to the increase in object count.
- There is more diversity in frame time among test scenes for higher intensity tests.

The graphs below provide a visualization of the frame time results presented in the tables within this section. When examining the graphs, keep in mind the differences in the x-axis scale. This is due to considerable variation in the range of values.
Figure 47: Graph of Test Results
The devices under test can be categorized based on their hardware specifications into high, medium, and low-end classes. The high-end category would include the LG G3, HTC One, and Samsung Galaxy Note 3, with the Acer Iconia A1 and Samsung Galaxy Tab 2 being medium range devices, and lastly the HTC One V being closer to low-end. This categorization seems to uphold the first observation: Devices with superior hardware specifications perform better in most scenarios. Each results graph shows that the high-end devices perform as well or better on the majority of individual and combined tests, with a few exceptions. The individual normal intensity tests graph shows that the HTC One exhibited noticeably weaker performance on the Lighting test in comparison to the HTC One V, which is on the low end of the device scale. This may be an indication of poorer fragment shader processing when generating the lighting effects on the HTC One. The other thing to notice is that the medium/low end devices struggle more with the Static and Lines tests, with the HTC One V not even being able to run the normal intensity Static test. Transitioning from normal to high intensity tests magnifies the weakness of all devices. The Collision test degradation isn’t unexpected since ten times more objects need to be computed. The collision detection algorithm is executed for every frame drawn and is CPU clock speed dependent. The Lines test becomes even more troublesome for the lesser devices with the Galaxy Tab 2 taking almost 500 milliseconds to draw one frame. Regardless of the intensity level of the test scenario, it is safe to conclude that a higher-end device will more frequently deliver better performance (stay under the 100 millisecond mark to render a frame) than a medium/low-end one.

The second observation: Degradation in performance is not proportional to the increase in object count, is the case for all test scenes when comparing normal to high
intensity results for individual tests. Examine the results for the Triangles (20k) normal intensity test and notice that the high intensity renders five times as many triangles but only yields a 3-fold increase in frame time on average across devices. This non-linear growth holds true for all the various tests. The one specific test result of note that is somewhat unique and deserves additional inspection is with the Texture test. This test shows only a minor difference of 3 milliseconds on average of added processing time between a 512x512 (normal) resolution texture and a 2048x2048 (high) resolution texture. Such a performance delta can be negligible given the resolution increase. It seems that the Tile-Based Renderer GPUs demonstrate their strong fragment processing capabilities with texturing. Another thing to consider is perhaps rendering a high-resolution texture at the same size as a lower-resolution one on the same screen has little effect on processing due to the same amount of pixels being filled on the screen. Android provides formats for texture compression but no compression was used in the test.

The last observation: There is more diversity in frame time among test scenes for higher intensity tests is made apparent with the visualization of the data with the bar graphs. The frame time’s standard deviation (averaged for all tests) for normal intensity tests is 8.7 in contrast to 85.6 for high intensity. The higher intensity tests are beneficial because they expose the weakness of a device with specific geometries. Having knowledge of which geometric primitives offer better performance is important information to possess when developing an OpenGL application. One of the outlying data points in the individual high intensity results is the poor performance of the Lines test. A Line primitive is constructed using two vertices, while in comparison a triangle primitive requires 3 vertices, but the scene consisting of 100,000 triangles shows significantly
better frame rate performance even though more vertex data needs to be processed. This finding holds true for all devices tested, making it safe to conclude that the use of line primitives should generally be kept to a minimum. Encountering unpredictable results like this further supports the claimed benefits of a customizable benchmarking solution that allows the user to push the device’s limits. Without being able to increase the intensity level of a test, drastic performance differences among some primitive types may not have been uncovered.

Based on the findings presented here, the general conclusion that can be made is that the use of higher resolution textures, fairly realistic lighting algorithms (fragment shader intensive work), and conservative use of any OpenGL primitive type to construct more complex objects, will result in stable and predictable performance on devices that fall into the medium through high-end range.

Antialiasing Results (All Devices)

The antialiasing test was run independently of the tests discussed in the results section above. A scene of 20,000 triangles was rendered to evaluate AA performance.

One of the advertised benefits of Tile-Based Renderer architectures is the minimal cost of applying antialiasing to a scene so a test was performed using the MOBILESpeed’s antialiasing feature on each device to evaluate this claim.
The following table presents test results (measured in frame time) from performing antialiasing on a scene containing 20,000 triangles. The purpose of this test was to evaluate the performance impact of adding antialiasing. As the results below show, the impact is minimal on the devices tested.

<table>
<thead>
<tr>
<th>Device</th>
<th>No AA</th>
<th>4x AA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acer Iconia A1</td>
<td>17.13 ms</td>
<td>17.14 ms</td>
</tr>
<tr>
<td>Samsung Galaxy Tab 2</td>
<td>29.85 ms</td>
<td>38.63 ms</td>
</tr>
<tr>
<td>HTC One V</td>
<td>21.34 ms</td>
<td>21.53 ms</td>
</tr>
<tr>
<td>Samsung Galaxy Note 3</td>
<td>17.01 ms</td>
<td>16.86 ms</td>
</tr>
<tr>
<td>HTC One</td>
<td>16.99 ms</td>
<td>16.97 ms</td>
</tr>
<tr>
<td>LG G3</td>
<td>No data available</td>
<td>No data available</td>
</tr>
</tbody>
</table>

Table 11: Frame Rate Comparison – No AA vs. 4x AA
FUTURE WORK

MOBILESpeed was designed to easily support integration of new benchmark tests or modification of existing ones. Given the time constraints for this study, a decision was made to prioritize the OpenGL ES benchmarks that were to be implemented by taking into account both the value of the benchmark and the implementation effort required. Due to the time constraints, not all tests were implemented in the current version of the product. A future update to MOBILESpeed will expand the scope of the application to include more advanced graphics rendering tests as well as modify some existing ones. The following is list of modifications to make to existing benchmark tests as well as additional benchmark tests to implement:

- **All tests** (modify):
  - Broaden the range of input values for the user to choose from when creating a custom scene.
  - Provide an option for off-screen rendering.

- **Texture test** (modify): provide the ability to render a variable number of textured cubes. Currently the number of cubes is fixed to three.

- **Lighting test** (modify): allow the user to customize the number of light sources in the scene (currently just one is allowed).

- **Collision test** (modify): implement collision detection using a Quadtree data structure for enhanced performance.

- **Multiple/complex fragment shader test** (add): switching between fragment shaders during the rendering of a frame is a suggested way to stress Tile-Based Renderers.
• **OpenGL driver overhead test** (add): this would be used to test and compare the performance of various OpenGL driver implementations.

Aside from benchmark related updates, the MOBILESpeed app can be enhanced by adding the capability to connect to an online database to store and retrieve test results. This database will contain a collection of devices, their hardware specs, and benchmark metrics, which can be used for quick device-to-device performance comparisons.

The application as it now is offers a fairly comprehensive set of benchmark tests and functionality, but a few more additions will make it suitable for publishing on the Google Play store.
CONCLUSION

This project was completed successfully and achieved its objective of creating a user-customizable mobile graphics benchmarking Android application called MOBILESpeed. The application is fully functional with an intuitive and user-friendly interface. It can be installed on any Android device that supports a minimum SDK version of 14 and OpenGL ES 2.0. MOBILESpeed features nine unique OpenGL benchmark scenes which can be run individually or combined into a single scene. Each benchmark scene allows the user to control the amount of objects to be rendered. It provides detailed frame rate results in terms of frame time and frames per second (FPS) at the conclusion of a benchmark test. It also provides a device information screen which displays to the user relevant hardware (CPU, memory, GPU, etc.) and software (API, library versions, etc.) information regarding the device being used.

The reader was educated on various topics in the field of graphics and then presented the design and implementation details of MOBILESpeed. The following topics relating to graphics were discussed:

- Graphics architecture on the mobile Android platform which provided perspective on the software layers involved in graphics processing.
- OpenGL ES 2.0 and its programmable processing pipeline which covered the stages of the pipeline and the role of vertex and fragment shader programs.
- GPU rendering architectures commonly implemented on mobile devices.
- A trade study of existing mobile graphics benchmarking applications.
- The concept of benchmarking which included explanations of frame rate, sources of performance bottlenecks, V-Sync, and antialiasing.
It was important to understand the topics mentioned above as they influenced the design and development decisions made for MOBILESpeed.

Development began after a set of requirements were written and software design was modeled in UML. The Eclipse integrated development environment with the Android Development Tools plugin was used to write and test the code. The code was written in Java to implement the application’s user interface and functionality while the graphics rendering was implemented using the OpenGL ES 2.0 library. The user interface of MOBILESpeed was designed to support Android devices of various screen sizes and resolutions. All project files were stored in a GitHub repository for version management.

Once development was complete, MOBILESpeed was deployed on multiple different Android devices and used to benchmark their performance. The performance data was gathered and presented in graphs and tables. The analysis of the results demonstrated the benefits of benchmarking with MOBILESpeed as several interesting and unexpected results were uncovered such as the relatively poor performance of the OpenGL lines primitive. The scene customizability feature allowed for all devices under test to be analyzed under intense scenarios, pushing the high-end ones to their limits.

Upon completion of the project, some notable lessons learned include:

- Utilize online resources, particularly Android and OpenGL ES tutorials to gain knowledge and overcome technical issues. Some of the better ones used for the development of this app can be found in the references section.

- For app development, use the Android Studio IDE instead of Eclipse as it provides an enhanced set of development and user interface design tools. At the
start of this project Android Studio was not yet the official development environment so the decision was made to go with the more stable Eclipse IDE.

- App layout design is crucial to its success. The various sizes and resolutions of devices on the market make it difficult to design a single user interface that consistently looks good across all devices. In most cases, multiple different UI layouts need to be created for each screen of the app to best support all devices.

- Test the application on actual Android hardware and not just using the emulator. The emulator is good for previewing the UI layout but is a resource hog and runs much slower than actual hardware. Make sure to test the application on hardware from various manufactures as each one may have a different OpenGL driver implementation which may lead to unexpected issues. Also, the emulator cannot test everything because some API calls require the presence of actual hardware otherwise an error occurs and terminates the app.

- OpenGL ES requires some additional steps to prepare vertex data for drawing, specifically creating a buffer of floating point values and converting them to use the device hardware’s native byte order. Be aware of this necessary step in the event that a desktop OpenGL application is being ported to a mobile app.

- Android’s OpenGL graphics rendering loop is a little different than a traditional one. All drawing is initiated through the onDrawFrame() method which is called by the system so all rendering code needs to conform to this structure.

- Attain a solid understanding of mobile GPU processing architecture prior to the design and development of any graphics benchmarking app.
This paper has established the foundation for creating a user-customizable graphics benchmarking Android application. It has provided design and development material, tools and techniques, and a comprehensive overview of benchmarking concepts and mobile graphics technologies.
REFERENCES

[1] Android Device Distribution
https://developer.android.com/about/dashboards/index.html

https://developer.android.com/design/get-started/principles.html

http://www.google.com/design/spec/material-design/introduction.html

http://community.imgtec.com/files/powervr-graphic-architecture-idc14-china-nov14/


[6] OpenGL Insights - Performance Tuning for Tile-Based Architectures
http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-TileBasedArchitectures.pdf

[7] Benchmark Definitions by Merriam-Webster
http://www.merriam-webster.com/dictionary/benchmark

[8] OpenGL Matrices
http://www.songho.ca/opengl/gl_projectionmatrix.html

[9] The OpenGL ES Shading Language

https://www.opengl.org/wiki/Performance

https://source.android.com/devices/graphics/architecture.html


https://developer.android.com/guide/topics/graphics/hardware-accel.html

[14] Android Views
[15] Android’s Surface Rendering Pipeline Diagram
https://source.android.com/devices/graphics/images/ape_fwk_graphics.png

[16] OpenGL ES on the Android Platform
http://developer.android.com/guide/topics/graphics/opengl.html

[17] OpenGL Wiki
https://www.opengl.org/wiki/OpenGL_ES

[18] Khronos OpenGL ES Guide
https://www.khronos.org/opengles/

[19] OpenGL ES Pipeline

[20] Rendering Pipeline Overview
https://www.opengl.org/wiki/Rendering_Pipeline_Overview

[21] GFXBench Benchmarking App
http://gfxbench.com/result.jsp

[22] 3DMark Benchmarking App
http://www.futuremark.com/benchmarks/3dmark

[23] Antialiasing Diagrams


[25] Understanding Rendering Techniques
APPENDIX A

Complete listing of UML class diagrams for the application.

UI Package Class Diagram

Model Package Class Diagram
Figure 50: Detailed Classes of the Model Package
Adapter Package Class Diagram

![Diagram of Adapter Package Class]

*Figure 51: Detailed Classes of the Adapter Package*

Util Package Class Diagram

![Diagram of Util Package Class]

*Figure 52: Detailed Classes of the Util Package*
Figure 53: Detailed Classes of the OpenGL Package - Part 1
Figure 54: Detailed Classes of the OpenGL Package - Part 2
Figure 55: Detailed Classes of the OpenGL Package - Part 3
Figure 56: Detailed Classes of the OpenGL Package - Part 4
Figure 57: Detailed Classes of the OpenGL Package - Part 5